Validador automático de prácticas usando Árboles de Comportamiento

Sistemas Informáticos 2011/2012

AUTOR
Rafael Antúnez Torrejón

DIRECTOR DE PROYECTO :
Marco Antonio Gómez Martín

Facultad de Informática
Universidad Complutense de Madrid

Documento maquetado con TEXIS v.1.0.

Validador automático de prácticas usando Árboles de Comportamiento

Facultad de Informática
Universidad Complutense de Madrid

Autorización

Se autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Rafael Antúnez Torrejón

Agradecimientos

En primer lugar queria darle las gracias a mi familia que tantas veces me ha apoyado a lo largo de la carrera dandome ánimos en tantos momentos.Tambien darle las gracias a Marco Antonio Gómez Martín por haber confiado en mi a la hora de hacer este proyecto y por la cantidad de ayuda que he recibido cuando tenía problemas a la hora de crear la aplicación.

Por último me gustaría darle las gracias a mis compañeos de prácticas de otras asignaturas a los que alguna vez he tenido que dejarles abandonados por mi dedicacion a este proyecto.

Resumen

En el presente documento se describe un sistema encargado de la evaluación de prácticas. La aplicación recibe como entrada los ficheros que forman la práctica y devuelve el resultado de comprobar si la práctica esta bien implementada o no.

Para ello, la aplicación primero lee los archivos de entrada (ficheros de la práctica creada por el alumno), los compila y los ejecuta. También recibe como entrada unos ficheros de test (los cuales le son proporcionados al alumno) los cuales tambien son compilados y ejecutados para ver si la práctica los admite.

A continuación, la aplicación coge los resultados devueltos por la ejecución del programa y los almacena en un fichero. Por último, se compara el contenido de este fichero con el contenido de un fichero con los resultados correctos que debería devolver la práctica en el caso de que estuviera bien implementada.

Para llevar a cabo este proceso, la aplicación hace uso de los denominados arboles de comportamiento. Los arboles de comportamiento son estructuras que se crean con la finalidad de organizar el comportamiento de un sistema y permiten trabajar con él de forma sencilla. Para ello, cada nodo del árbol representa una tarea junto a su condición de ejecución.

En nuestro caso tendríamos, entre otros, nodos para compilar, para ejecutar y para comparar ficheros.

Palabras Clave evaluador de prácticas, validador automático, árbol de comportamiento, Java

Abstract

This document describes a system in charge of practice assessments. The application receives as input the files that make up the practice and returns the result to check whether the practice is either implemented or not.

This application first reads the input files (practice files created by the user), compiles and executes them. It also receives as input some test files (which are provided to the student) which are also compiled and executed to see whether the practice support them or not. Then, the application takes the results returned by the execution of the program and stores them in a file. Finally, it compares the content of this file with the contents of the file that contains the correct results that should return the practice if it was well implemented.

To carry out this process, the application makes use of so-called behaviour trees. Behaviour Trees are structures that are created with the purpose of organizing the behaviour of a system and allow to work with it easily. In order oo do that, each tree node represents a task with its execution condition. In our case we would have, among other, nodes to build, to run and to compare files.

Key words practices evaluator, automatic validator, behaviour trees, Java

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

Resumen: En este capitulo introduciremos de que va el proyecto desarrollado. En la secci´on 1.1 hablaremos de los motivos que han llevado al desarrollo del proyecto. En la sección 1.2 hablaremos de la solución creada para resolver el problema del que se habla en la sección 1.1. Por último, en la sección 1.3, hablaremos de la distintos capítulos en los que se divide este documento y describiremos de que trata cada uno.

1.1. Motivación

El principal objetivo de la aplicación desarrollada es el de hacerle la vida más fácil tanto a los estudiantes como a los profesores en cuanto a entregas de prácticas se refiere.

Para ver como les hace la vida más fácil, vamos a ver 2 situaciones. En la primera describimos el caso en el que el profesor ni el alumno usan nuestra aplicación. En la otra situación vamos a ver que pasaría en el caso de que si que la usarán.

1.1.1. La aplicación no se usa

El profesor les pone una práctica a los alumnos y les da el enunciado. Los alumnos realizan la práctica y, para probarla, meten como valores de entrada algunos ejemplos, aunque la mayoría de las veces los ejemplos que usan son aquellos para los que la práctica no suele fallar. Entonces, los alumnos creen que la práctica la han implementado correctamente y se la muestran al profesor personalmente o se la pasan para que el profesor la miré con más detenimiento.

En el caso de que los alumnos tuvieran que enseñarle personalmente la práctica al profesor (aunque tuviera que entregársela después), hay muchas veces que se encuentran en la situación en la que el profesor usa como ejemplos de prueba aquellos que pueden hacer que la práctica falle o haga cosas raras. Entonces en ese momento, los alumnos se quedan sorprendidos y empiezan a pensar que el profesor ha usado ese ejemplo de entrada para pillarles en la práctica y suspenderles.

Después de esto puede darse el caso de que los alumnos tengan que corregir la práctica hasta que los ejemplos del profesor no fallen, o que el profesor les diga que no pasa nada y que pueden entregarla aunque la nota no vaya a ser perfecta.

En el caso de que los alumnos tuvieran que entregarle la práctica al profesor y éste la tuviera que corregir más tarde sin la presencia de los alumnos, el profesor tendría que gastar mucho tiempo probando muchos ejemplos para cada práctica entregada por los alumnos. Como se puede suponer, esto puede ser un poco tedioso para el profesor y bastante molesto por los alumnos. Esto último se puede deber a que cuando el profesor les da la nota de la práctica a los alumnos, algunos podrían llegar a quejarse en el caso de que la nota fuera muy baja comparada con la que ellos pensaban, alegando que ellos habían hecho muchas pruebas y todas pasaban correctamente. Esto último haría que los alumnos le pidierán una tutoría al profesor para que les explicarán el porqué de la nota, por lo que el profesor tendría que buscar un hueco en su horario para atenderles. Si solamente se queja un grupo no pasa nada, pero si se quejan varios, el profesor tendría que buscar muchos ratos libres para atenderles lo que le puede costar mucho si tiene otras cosas que hacer.

1.1.2. Usando la aplicación

El profesor les pone una práctica a sus alumnos y les pasa tanto el validador como varios ficheros de prueba y tests. Además también les pasa un fichero con las soluciones que les tendría que dar la práctica en el que estuviera bien implementada. Los alumnos implementan la práctica y para probarla, le pasan el validador que les ha pasado el profesor. Si el validador no se queja significa que han implementado bien la práctica y que el profesor les va a poner buena nota. En el caso contrario, sabrán que tienen que corregirla ya que si la entregan tal y como la tienen el profesor les va a poner mala nota. De esta manera, cuando el profesor les ponga a los alumnos las notas, no les pillará a ellos de improviso. Además,el profesor, cuando reciba las prácticas de los alumnos, ya sabe que funcionan correctamente (a no ser que haya algun alumno listillo que entregue la práctica sin pasarle las pruebas adjuntadas junto al validador).

La principal desventaja es que el rofesor debe invertir tiempo en construir

el validador para la correspondiente práctica.

1.1.3. Conclusiones

De todo lo explicado anteriormente, se puede concluir que gracias a nuestra aplicación los profesores no tendrán que perder el tiempo corrigiendo todas las prácticas a todos sus alumnos ni estos últimos se llevarán una sorpresa cuando el profesor les entregue la nota de la practica.

1.2. Solución

Para llevar a cabo la aplicación descrita anteriormente, se ha creado un framework (o conjunto de clases) que atyudan en la construcción del validador de prácticas.

El trabajo desarrollado se divide en 3 partes:

1. **Framework EagerBT** : En esta primera parte se desarrolla el framework encargado de la ejecución de árboles de comportamiento. En ella, se crean los nodos básicos sobre los que se construyen los árboles de comportamiento.

2. **Framework validación** : En esta parte se desarrolla el framework encargado de la ejecución y compilación de código fuente.

3. **Validador** : En esta parte se crea la aplicación encargada de la validación de prácticas aplicandola sobre una en particular.

1.3. Estructura del documento

Este documento esta dividido en los siguientes capítulos:

- **Estado del arte** En este capítulo se hablará de la historia de los árboles de comportamiento y se describirá otras aplicaciones en las que se ha usado. Ádemas hablaremos de otros validadores de prácticas que han sido creados por terceros.

- **EagerBT** En este capítulo se presenta EagerBT, el framework implementado para ejecutar árboles de comportamiento. Se hará un recorrido por todos los diferentes nodos básicos con los que se puede formar un árbol de comportamiento. También se incluirán varias pruebas que han servido para demostrar el correcto funcionamiento de cada uno.

- **Framework de validación** En este capítulo ampliaremos el repertorio de nodos incluyendo nodos más complejos como los dedicados a la

compilación y ejecución de código que son necesarios para construir validadores.

- **Casos de uso** En este capítulo se tratará más en profundidad el uso de esta aplicación usándola para validar una práctica real.
- **Trabajo futuro y conclusiones :** En este capítulo se hablará de las posibles ampliaciones del trabajo aquí presentado. También se hablará de las conclusiones a las que se han llegado después de desarrollar el proyecto.

Capítulo 2

Estado del arte

Resumen: En la sección 2.1 empezaremos hablando de los árboles de comportamiento y su uso en algunas aplicaciones. A continuación, en la sección 2.2, hablaremos de JBT un framework de Java el cuál ayuda a construir y ejecutar árboles de comportamiento de una forma sencilla. Para terminar, en la sección 2.3, hablaremos de otro validador de prácticas.

2.1. Árboles de comportamiento

2.1.1. Definición de árboles de comportamiento

Los árboles de comportamiento son estructuras que se crean con la finalidad de organizar el comportamiento de un sistema y permitir trabajar con él de forma sencilla. EstÃ¡n formados por elementos más básicos llamados `Nodos` que son los que definen las acciones que definen el comportamiento del árbol.

Saber a que parte del código hay que ir para modificar ciertos comportamientos y cuando invocarla es una tarea que se hace mucho más rápida utilizando estructuras como ésta.

2.1.2. Creación de los árboles de comportamiento

Para la creación de un árbol de comportamiento se deben seguir los siguientes pasos:

1. Pensar en el comportamiento que se quiere que tenga el árbol.
2. Pensar en los pasos de ejecución del árbol de comportamiento.

3. Pensar en los nodos que se necesiten para crear dicho comportamiento.
4. Implementar dichos nodos.
5. Unir todos los nodos para formar el árbol de comportamiento.

Los árboles de comportamiento que se pueden crear mediante nuestro frameworks tienen las siguientes restricciones:

- Los nodos encargados de la implementación de condiciones solo pueden aparecer como nodos hoja.
- No se permiten combinar nodos de condiciones y de acciones básicos en un mismo nodo. Los únicos nodos con los que se pueden combinar son con los nodos que implementan estructuras de control

En los árboles de comportamiento generales (los que se usan en campos como la Inteligencia Artificial) se permite que los nodos del árbol puedan realizar cualquier acción sin restricción ninguna ya que se permite, por ejemplo, que un nodo esté formado por una condición y por un conjunto de instrucciones las cuáles se ejecutarán si se cumple la condición. Debido a esto, el tamaño que puede tener un árbol de comportamiento puede ser más pequeño que un árbol creado mediante nuestro framework y que se comporte igual.

Imaginemos que queremos crear un árbol que muestre un mensaje si un número es positivo. En la siguiente figura podemos ver como sería la estructura del árbol cuando se usan los nodos de nuestro framework (árbol de la izquierda) y cuando se usan los árboles de comportamiento generales(árbol de la derecha).

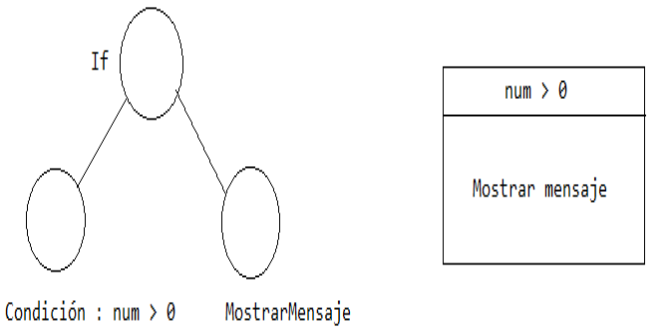


Figura 2.1: Diferencia entre árboles

Como se puede observar, el árbol de comportamiento es más grande si se usan los nodos de nuestro framework (3 nodos frente a 1).

2.1.3. Ejecución de árboles de comportamiento

Tanto los árboles de comportamiento de nuestro framework como los que se usan en campos como la Inteligencia Artificial se ejecutan de forma parecida. Ambos empiezan desde un nodo raíz y van ejecutando los nodos hijo en un orden que depende del tipo del nodo en el que nos encontremos.

La diferencia radica en que mientras que los árboles de nuestro framework se ejecutan de golpe (de una sola vez), los otros árboles se pueden ir ejecutando poco a poco.

2.1.4. Donde se usan los árboles de comportamiento

Algunos de los juegos que actualmente están en el mercado utilizando estos sistemas son el Grand Theft Auto, el Halo o Spore. Todos ellos se basan en el uso de estas estructuras para gestionar comportamientos y son referentes en sus respectivos campos.

Figura 2.2: Halo

A diferencia de los árboles de comportamiento que vamos a crear nosotros, los árboles que se usan en juegos como los mencionados anteriormente poseen nodos internos de tipo `Condición` que ademÃ¡s contienen instrucciones a ejecutar.

En la figura 2.5 se muestra un ejemplo de estos árboles.

Por ejemplo, la condición puede ser (Si VeoEnemigo == 1). La lista de acciones sería (Alertar,Esconderse).

Como medida de simplificación, las acciones las realizarán generalmente los nodos "hoja", aunque para comportamientos muy bien definidos de los que se puede extraer características comunes podremos lanzar acciones desde nodos internos. Por ejemplo al controlar el comportamiento de la ca-

Figura 2.3: Grand Theft Auto IV

Figura 2.4: Spore

beza de un soldado podemos querer tener un "supercomportamiento" como el parpadeo ocular, independiente del estado en el que nos encontremos (esto significa no que adquiera superpoderes sino que el comportamiento se manifieste de forma común a los nodos descendientes que pueda tener). Los hijos pueden ser "patrullando" y "siguiendo objetivo". En el primero estableceremos un comportamiento que haga rotar la cabeza desde el frente a -45 y 45 grados respecto al plano vertical. El segundo podemos definir que la cabeza se oriente a un determinado objetivo. De forma independiente se verá el parpadeo de los ojos que es común a los dos. Se comenta que el 99 % de los nodos con acciones serán nodos hoja por lo que será bueno tenerlo en cuenta.

Al mismo tiempo que el árbol anterior, podríamos tener uno trabajando en paralelo para controlar el movimiento de la boca o incluso la expresión facial. Ya a partir de este punto empezamos a ver la utilidad que tienen y la facilidad que va a tener añadir nuevos comportamientos, tanto adicionales a los que tenemos como totalmente nuevos.

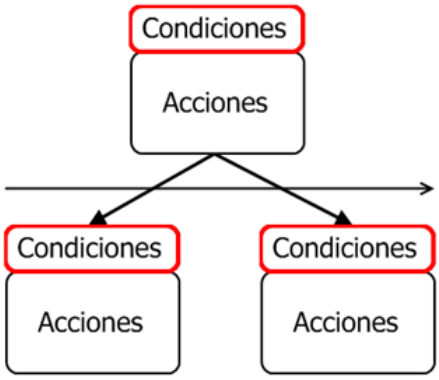


Figura 2.5: Ejemplo de árbol de comportamiento de juegos

Estos modelos se utilizan para hacer más sencilla la inclusión de nuevos comportamientos que añadan complejidad a las respuestas de los agentes inteligentes, simplificar el trabajo de creación de los propios comportamientos, permitir el prototipado y la puesta en escena de nuevas iteraciones y optimizar el rendimiento del sistema.

2.2. JBT (Java Behaviour Trees)

2.2.1. Introducción

JBT en un framework de Java para construir y ejecutar árboles de comportamiento (BT : Behaviour Trees). En los últimos años los BTs han sido extensamente aceptados como una herramienta para definir el comportamiento de los personajes de los videojuegos. Sin embargo, no existe una implementación de software libre Java de tal tecnología. Con JBT se intenta proporcionar un software sólido para crear y ejecutar BTs.

JBT es una librería gratuita para Eclipse JDK. Tiene 2 partes:

- **JBT Core :** Parte que implementa todas las clases necesarias para crear y ejecutar BTs. Básicamente, JBT Core permite al usuario crear BTs en Java puro y ejecutarlos. Con el fin de aliviar la tarea de creación de los BTs, JBT Core incluye varias herramientas que automatizan el proceso de creacción de los BTs. En particular, permite crear un BT desde un fichero XML que contiene la descripción. Haciendo esto, el usuario de este framework solo se tiene que preocupar de definir los BTs en archivos XML e implementar las acciones de bajo nivel y condiciones que va a usar su árbol, las cuáles son dependientes del dominio.

- **JBT Editor :** Es una interfaz gráfica de usuario que puede ser usada para la definición de BTs y posteriormente para exportarlas a un fichero XML en el formato que entiende el JBT Core. JBT Editor proporciona un conjunto de nodos estándar para la creacción de los BTs. Incluye nodos tales como secuenciales, paralelos, decoradores, etc.

Los BTs creados por el JBT son dirigidos por ticks, lo que significa que para que tengan tiempo de CPU, necesitan ser marcados externamente. Siguiendo este patrón, el usuario podrá controlar cuanto tiempo de CPU va a consumir el BT

2.2.2. Arquitectura de JBT

A continuación vamos a explicar un poco de la arquitectura de JBT.

Modelo conducido por ticks

Como contamos anteriormente, JBT implementa un modelo BT dirigido por ticks. Un BT debe ser evaluado a través de marcas, así que en cada ciclo un agente externo marca el árbol para que este actualice su estado. Un tick es sólo una manera de darle al árbol cierto tiempo de CPU para que actualice su estado; en particular, los tick son usados para darles a los nodos del árbol algo de tiempo para evaluar si han acabado o no, y consecuentemente hacer que evolucione el árbol.

El enfoque más simple a los BTs dirigidos por ticks es el de marcar al nodo raíz y entonces permitir que cada nodo marque recursivamente a sus nodos hijos. Pero este es un proceso muy ineficiente, ya que muchos nodos están esperando a que sus nodos hijo acaben. Por lo tanto no deberián recibir ticks, desde que al menos que sus hijos hayan acabado, no harán nada útil cuando reciban el tick. Por lo tanto, en general, muy pocos nodos deberían ser marcados en un ciclo, y como resultado JBT implementa una lista donde solo están almacenados los nodos que tienen que ser marcados.

Módelos independientes de la ejecución

Cuando se ejecuta un BT, debería haber una clara distinción entre el árbol que está siendo ejecutado (el modelo) y como está siendo ejecutado actualmente (la ejecución).

Por cada comportamiento particular, distinguimos entre el modelo BT que lo define y como está siendo ejecutado. El *como* es lo que hace el *BT Executor*. Por cada entidad que quiere ejecutar un comportamiento (Model BT) existe un BT Executor el cuál coge el Modelo BT y lo procesa (sin modificarlo) simulando el comportamiento representado por el Modelo BT.

Esta elección implica que, aparte del Modelo BT, hay otro tipo de árbol, el Executor BT. Cuando una entidad quiere ejecutar un comportamiento, el BT Executor coge el Modelo BT y crea un Executor BT para ejecutar el comportamiento.

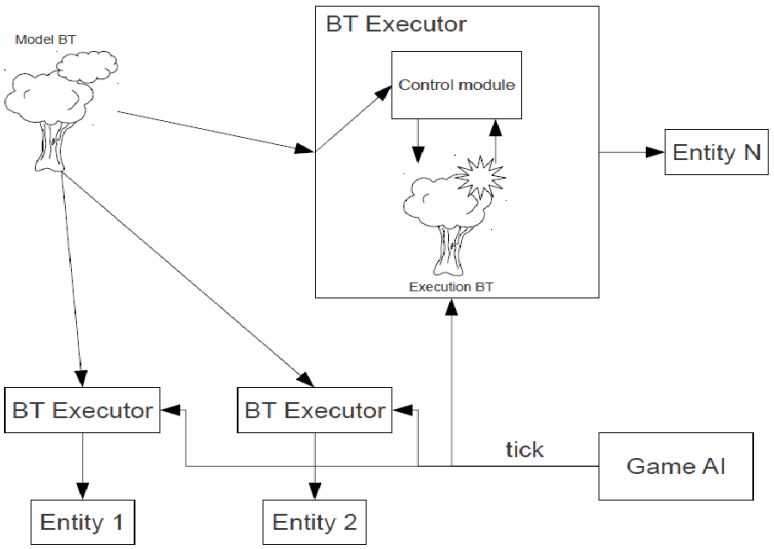


Figura 2.6: Visión general de la arquitectura de los BT.

Arquitectura

La figura 2.6 muestra una visión general de la arquitectura del JBT Core.

Hay un Modelo BT que representa un comportamiento particular. También hay un BT Executor por cada entidad que quiere ejecutar el Modelo BT. Cada BT Executor hace uso del Modelo BT y construye un BT Execution que actualmente ejecuta el comportamiento conceptualizado por el Modelo BT.

El usuario de este framework no tiene que saber todos los detalles de como trabaja JBT internamente, pero como tiene que implementar algunas clases para ejecutar sus propios arboles al menos debería conocer su arquitectura general.

Modelo BT

Antes de describir más en detenimiento el Modelo BT vamos a ver un ejemplo simple de árbol de comportamiento.

El árbol en la figura 2.7 representa un árbol que es usado por el personaje de un juego que quiere abrir una puerta. Lo primero de todo, comprueba si la puerta está cerrada (condición DoorClosed). Si lo está, entonces intenta

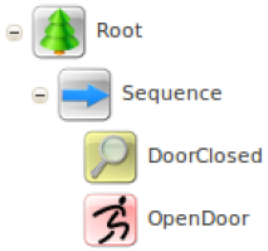


Figura 2.7: Un simple árbol de comportamiento.

abrir la puerta ejecutando la acción OpenDoor.

En este árbol se pueden ver 4 nodos. El nodo llamado *Root* es solo la raíz del árbol y no tiene otro significado aparte de ese. Entonces hay un nodo *Sequence* el cuál ejecuta secuencialmente sus 2 nodos hijos (la condición *DoorClosed y la acción OpenDoor*). El nodo *Sequence* es un nodo estándar pero tanto el nodo *DoorClosed* como el nodo *OpenDoor* son dependientes del dominio es decir, han sido definidos por el usuario para que tengan un significado útil dentro del juego que se está jugando.

Todos los nodos en un BT tienen un *contexto de ejecución* es cuál es normalmente compartido por todos ellos. El contexto de ejecución , o *contexto* a secas, actúa como una pizarra que puede ser usada por los nodos para escribir y leer variables.

Sin embargo, no siempre todos los nodos comparten el mismo contexto. En general, el contexto de un BT es pasado hacia abajo de los nodos padre a los nodos hijos. Así, el contexto es el del nodo raíz, el cuál lo pasará a sus nodos hijos. Los hijos del nodo raíz pasarán su contexto a los nodos hijos, estos a sus hijos, y así hasta los nodos hoja. Sin embargo, algunos nodos no pasan su contexto a sus nodos hijos, sino otro distinto. Este nuevo contexto puede estar vacío o no, o puede ser de otro tipo.

JBT soporta los siguientes tipos de contextos:

- **Contexto básico :** Este es un contexto normal sin características especiales. Es el contexto que puede crear el usuario del framework.
- **Contexto jerárquico :** Un Contexto jerárquico tiene otro contexto (*el contexto de entrada*) como base. Cuando el contexto jerárquico no puede encontrar una variable dentro del conjunto de sus variables, le preguntará a su *contexto de entrada* por la variable.
- **Contexto seguro :** Un contexto seguro tiene otro contexto (*el contexto de entrada*) como base. Inicialmente todas las variables se leen

del contexto de entrada. Sin embargo, cuando una variable es modificada, su valor no es modificado en el contexto de entrada, en su lugar es modificado localmente. A partir de entonces, la variable se leerá localmente (es decir, desde el conjunto de variables del Contexto seguro) en lugar de leerlas del contexto de entrada.

- **Contexto seguro de salida :** Se comporta de la misma manera que el Contexto seguro. Contiene otro contexto, el *contexto de entrada* como base. Sin embargo, este contexto también contiene un conjunto de variables de salida (el cuál es una lista de nombres de variables). Esta lista representa las variables que pueden ser modificadas en el contexto de entrada. Las otras variables serán almacenadas localmente en el conjunto de variables del Contexto Seguro de Salida. Así, cuando el Contexto Seguro de Salida modifica el valor de una variable, normalmente establece su valor en una variable local (es decir, una variable proveniente del Contexto Seguro de Salida). Sin embargo, si la variable es una de las de la lista de las variables de salida, el valor se establecerá en el contexto de entrada el cuál será modificado. Cuando se recuperan variables, una variable de la lista de variables de salida siempre será recuperada del contexto de entrada. Una variable que no esté en la lista, será recuperada del contexto de entrada lo que pasa es que, cuando sea modificada, el valor será recuperado del Contexto Seguro de Salida (es decir desde el momento en el que una variable que no esté en la lista sea modificada, es gestionada localmente).

JBT ofrece un gran rango de tareas que pueden ser usadas para construir árboles de comportamiento.
Alguna de las tareas que soporta JBT son las siguientes:

- **Nodo Secuencial :** Nodo que ejecuta secuencialmente todos sus nodos hijo en orden. Si un nodo hijo falla, el nodo falla. Si todos los nodos hijo terminan con éxito, el nodo tiene éxito.

- **Nodo Selección :** Nodo que ejecuta secuencialmente todos sus nodos hijo en orden. Si un nodo hijo tiene éxito, el nodo tiene éxito. Si todos los nodos hijo fallan, el nodo falla.

- **Nodo paralelo :** Nodo que ejecuta concurrentemente todos sus nodos hijo. Este nodo tiene una *política de paralelismo*Si la política es *secuencial*, el nodo falla si un nodo hijo falla y tiene éxito si todos los nodos hijo tienen éxito. Si la política es de selección, el nodo falla si todos los nodos hijo fallan y tiene éxito si algún nodo hijo tiene éxito.

- **Nodo Selección Aleatoria :** Nodo que ejecuta todos los nodos hijo

en orden aleatorio. El nodo falla si algún nodo hijo falla y tiene éxito si todos los nodos hijo tienen éxito.

- **Nodo Secuencial Aleatorio :** Nodo que ejecuta secuencialmente todos los nodos hijo en orden aleatorio. El nodo tiene éxito si uno de los nodos hijo tiene éxito y falla si todos los nodos hijo fallan.
- **Nodo Inversor :** Nodo que invierte el código de estado devuelto por su nodo hijo.
- **Nodo Límite :** Nodo que limita el número de veces que puede ser ejecutado un nodo. El nodo falla si se ha superado el límite.
- **Nodo Repetición :** Nodo que ejecuta su nodo hijo indefinidamente.
- **Nodo Hasta Fallo :** Nodo que ejecuta su nodo hijo hasta que este falle.
- **Nodo Exitoso :** Nodo que tiene siempre éxito sin importar si el nodo hijo tiene éxito o falla.
- **Nodo Gestión Contexto Jerárquico :** Nodo que crea un nuevo contexto para su nodo hijo. El contexto que crea es un Contexto Jerárquico cuyo contexto de entrada es el contexto que se le pasa a este nodo y está vacío (sin variables).
- **Nodo Gestión Contexto Seguro :** Lo mismo que el anterior pero se crea un Contexto Seguro.
- **Nodo Gestión Contexto de Salida Seguro :** Lo mismo que el anterior pero se crea un Contexto de Salida Seguro.
- **Nodo Espera :** Nodo que se sigue ejecutando durante un periodo de tiempo y entonces tiene éxito. El periodo de tiempo puede ser especificado por el usuario.
- **Nodo Renombramiento :** Nodo que renombra una variable del contexto.
- **Nodo Éxito:** Nodo que acaba con éxito inmediatamente.
- **Nodo 'Fallo:** Nodo que falla inmediatamente.

Pasos JBT

Los pasos que hay que seguir para crear y ejecutar un árbol de comportamiento en JBT son los siguientes:

- **PASO 1 :** Definir las acciones de bajo nivel y las condiciones.
- **PASO 2 :** Implementar las acciones de bajo nivel y las acciones.
- **PASO 3 :** Crear los BTs con JBT Editor.
- **PASO 4 :** Crear una declaración Java de los BTs.
- **PASO 5 :** Ejecutar los árboles de comportamiento.

2.3. Otros validadores de prácticas

En esta sección describiremos un validador de prácticas llamado eGrader el cuál ha sido creado en la Universidad de Sharjah en los Emiratos Arabes.

2.3.1. Introducción

EGrader es un sistema de calificación basado en grafos. Es un sistema para calificar las soluciones Java de los alumnos, estáticamente y dinámicamente, en cursos de introducción a la programación. Los informes generados por eGrader hacen de él un sistema único no solo para las presentaciones de las notas de los estudiantes y para proporcionarles detalles sino también para asistir a los profesores para la creación de una base de datos con los trabajos de los alumnos y producir documentos como resultados de salida. Además, eGreader es uno de los pocos sistemas que soporta código Java con errores semánticos.

2.3.2. Metodología

EGrader puede calificar eficientemente y exactamente un código fuente en Java usando análisis estáticos y dinámicos. El análisis dinámico se lleva a cabo usando el framework JUnit. El análisis estático consta de 2 partes : la semejanza estructural basada en la representación en grafos del programa y la calidad medida por las métricas del software. La representación mediante grafos está basada en el Grafos de Control de Dependencias (GCD) y las Dependencias de Llamadas a Métodos (DLM) construidos a partir de los árboles de sintaxis abstractos del código fuente. Desde la representación en grafos, la estructura y las métricas de software son especificadas junto a la posición de las estructuras de control y representadas como un código al que llamamos *patrón de identificación*. Los patrones de identificación para las soluciones de los modelos son generadas en la primera fase. Los patrones de identificación para la entrega de los estudiantes son producidas en la segunda fase. El resultado del análisis estático es el resultado del proceso de comparación entre los patrones de identificación de los estudiantes y los

patrones de identificación de los modelos en la tercera fase. Estas 3 fases están representadas en la figura 2.8:

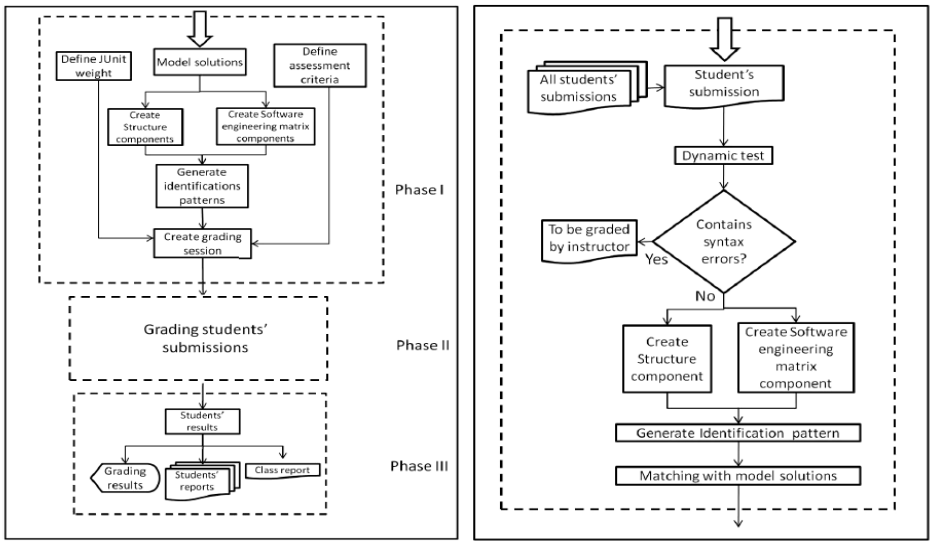


Figura 2.8: Las 3 fases del eGrader (izquierda) y la fase 2 del eGrader (derecha)

Patrón de identificación

El patrón de identificación es una representación de la estructura y de las métricas de ingenierÃa del software de un programa. La estructura está representada en el patrón de identificación basado en la traza del código (sin ejecutarlo) comenzando en el método main().
En la figura 2.9 se muestra un ejemplo de patrón de identificación.

El componente de estructura consta de varios sub componentes representados con una máscara de dígitos. Cada sub-componente representa una estructura de control o una llamada a método en la estructura del programa. Cada sub-componente está compuesto de 3 tipos de codificación : categoría básica, control y posición.

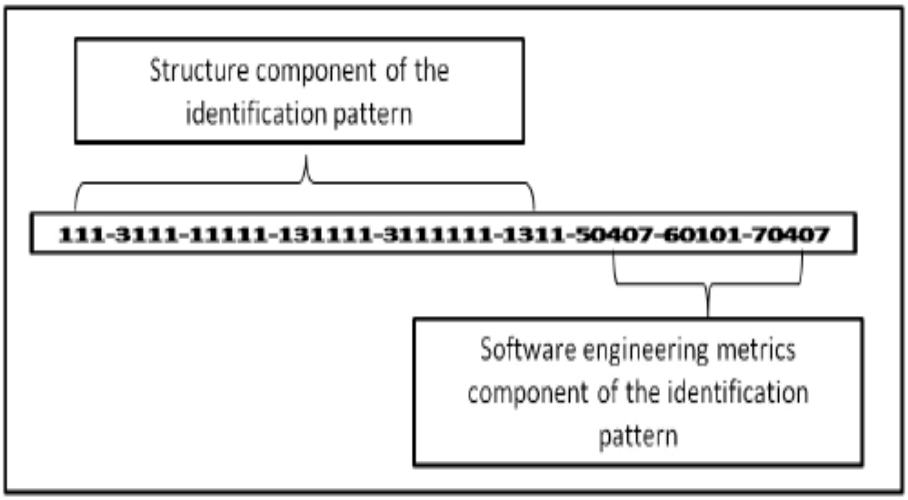


Figura 2.9: Ejemplo de patrón de identificación

En la tablas 2.1 se muestra la representación en código de las categorías básicas y controles de los componentes de estructura :

Los códigos de posición constan de 1 o más dígitos representando la posición de una estructura de control o de una llamada a un método en la estructura completa del programa. También representa la posición relativa a otra estructura de control o llamada a método dentro de la estructura del programa.

La figura 2.10 muestra un ejemplo de elemento de estructura para el patrón de identificación Compute Factorial. La clase Compute Factorial llama al método factorial para calcular el valor factorial después de comprobar su validez (número >= 0). Para hacer la traza de Compute Factorial, empezamos con la estructura de control en la línea 24 (si (número >= 0)). Desde que esta estructura de control es un control de condición del tipo if, tanto la categoría básica como el control se establecen a 1. La posición de esta estructura de control se establece a 1 al ser la primera estructura de control para trazar. La segunda estructura de control para trazar es la llamada al método en la línea 26 (fact = factorial(number)) la cuál es una llamada a un método no recursivo.

Categoría básica	Código	Estructura de control	Código
Condiciones	1	declaración if	1
Condiciones	1	declaración elseif	2
Condiciones	1	declaración else	3
Condiciones	1	declaración switch	4
Condiciones	1	declaración case	5
Condiciones	1	declaración general	*
Bucles	2	bucle for	1
Bucles	2	bucle while	2
Bucles	2	bucle dowhile	3
Bucles	2	bucle general	*
Llamadas a métodos	3	llamada recursiva	1
Llamadas a métodos	3	llamada no recursiva	2
Llamadas a métodos	3	llamada general	*
Excepciones	4	bloque try	1
Excepciones	4	bloque catch	2
Excepciones	4	bloque finally	3
Excepciones	4	bloque general	*

Tabla 2.1: Categorías básicas y Controles de los componentes de estructura de los patrones de identificación

Basándonos en la tabla, la categoría básica para la llamada al método es 3 y un método no recursivo tiene código de control 2, Desde que fact = factorial(number) es dependiente de control en la primera estructura de control para trazar, la cuál es if(number >= 0), el número de dígitos en el código de posición incrementará en 1 y será 11. La instrucción if en la línea 38 dentro del método factorial tiene código 11111, donde el primer 1 es por la categoría básica (condiciones) el segundo 1 es por el control (if) y el restante 111 es por la posición porque es dependiente de la declaración en la línea 15. Continuamos haciendo esto hasta terminar con todo el código. El componente de estructura completo ordenado del patrón de identificación de ComputeFactorial es el siguiente : 111-3211-11111-22112-1311.

SEM (Software Enigineering Metrics) consta de 3 sub-componentes : número de variables, número de clases y números y llamadas a métodos incorporados. Cada sub-componente consta de 2 o 3 partes dependiendo si el componente SEM es para el programa de un estudiante o para un programa modelo.

Para el programa de un estudiante, cada sub-componente consta de 2 partes : Categoría Básica y Número. Los códigos de categorías básicas son 5 para las variables, 6 para las clases y 7 para las llamadas a métodos de

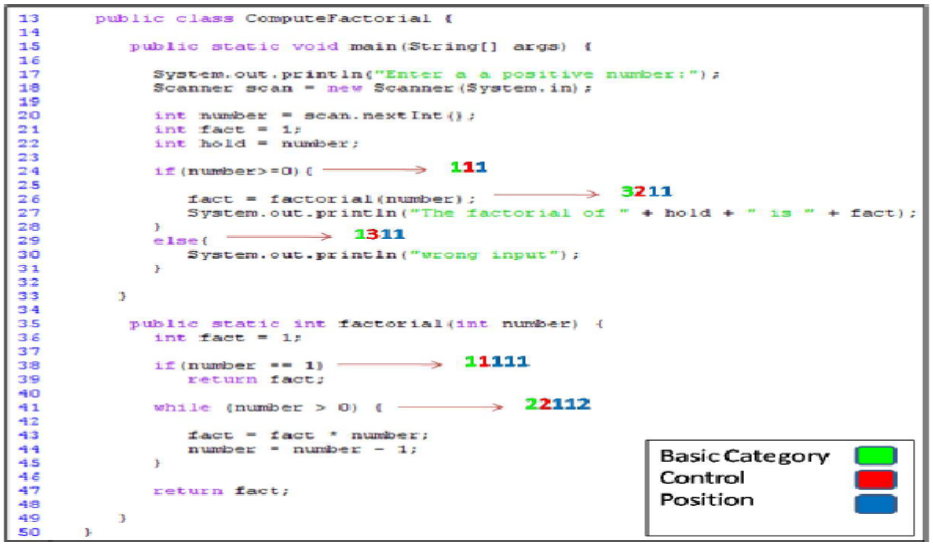


Figura 2.10: Clase ComputeFactorial

librerías. El Número representa el número de cada componente SEM en el programa del estudiante.

Para el programa modelo, cada sub-componente consta de 3 partes : La Categoría Básica, NúmeroMínimo y NúmeroMáximo. El código de categoría básico sigue las mismas reglas que el del programa de estudiante. Los parámetros NúmeroMínimo y NúmeroMáximo constan de 2 dígitos representando cada uno el mínimo y el máximo número de componentes SEM permitidos, respectivamente.

El componente SEM del programa de estudiante de la figura 2.10 es 507-601-704 que es calculado de la siguiente manera : 507 porque el estudiante usa 7 variables (5 = Variable), 601 porque hay 1 clase (6 = clase) y 704 porque hay 4 llamadas a métodos de librerías (7 : llamadas a métodos de librerías).

Si el programa es un programa modelo entonces el componente SEM es 50407-60101-70410 lo que significa que al estudiante solo se le permite usar entre 4 y 7 variables, solamente una clase y entre 4 y 10 llamadas a métodos de librerías.

La idea principal detrás del patrón de identificación es analizar tanto la estructura como las SEM del programa del estudiante. Por lo tanto, se requiere una estrategia eficiente para comparar los patrones de identificación. Se necesitan conocer ciertos criterios para desarrollar una estrategia eficiente para comparar patrones de identificación. Dichos criterios son los siguientes:

1. La coincidencia de patrones de identificación están basados en la distancia entre ellos. La medición de la distancia está definida como el número de estructuras de control y componentes SEM que faltan en el programa modelo además del número de estructuras de control extras y componentes SEM en el patrón de identificación del estudiante. Formalmente, la distancia es igual al valor absoluto de la suma del número de estructuras de control que faltan y el número de estructuras de control extras.

2. Si existe un patrón de identificación modelo que coincide exactamente con el patrón de identificación del estudiante, la distancia es igual a 0.

3. Si no se encuentran coincidencias exactas, la mejor coincidencia es el patrón de identificación del modelo el cuál tiene la distancia mínima al patrón de identificación del estudiante.

4. Si los 2 patrones de identificación de los modelos tienen la misma distancia al patrón de identificación del estudiante, la mejor coincidencia es la que maximiza la nota de calificación.

Para ilustrar este proceso de comparación, veamos un ejemplo para calcular el factorial de un número. Este ejemplo consta de 2 soluciones modelo y una solución de un estudiante. La primera solución modelo calcula el factorial usando un método recursivo (figura 2.11). La segunda es una solución no recursiva. Un ejemplo de la solución de un estudiante se muestra en la figura 2.12.

El patrón de identificación del estudiante es comparado con el patrón de identificación del primer modelo en la figura 2.13. La categoría básica y el control de cada estructura de control del patrón de identificación del estudiante es comparado con la categoría básica y el control de cada estructura de control del patrón de identificación del modelo hasta que se encuentra una coincidencia. La distancia en este ejemplo es igual a 2 ya que faltan 2 estructuras de control : un if y un elseif.

El patrón de identificación del estudiante también es comparado con el patrón de identificación del modelo para la solución no recursiva. El resultado de la comparación es el siguiente : 2 estructuras de control extras, 2 estructuras de control que faltan y un SEM que falta. Por lo tanto, la distancia es 6.

Figura 2.11: Solución recursiva

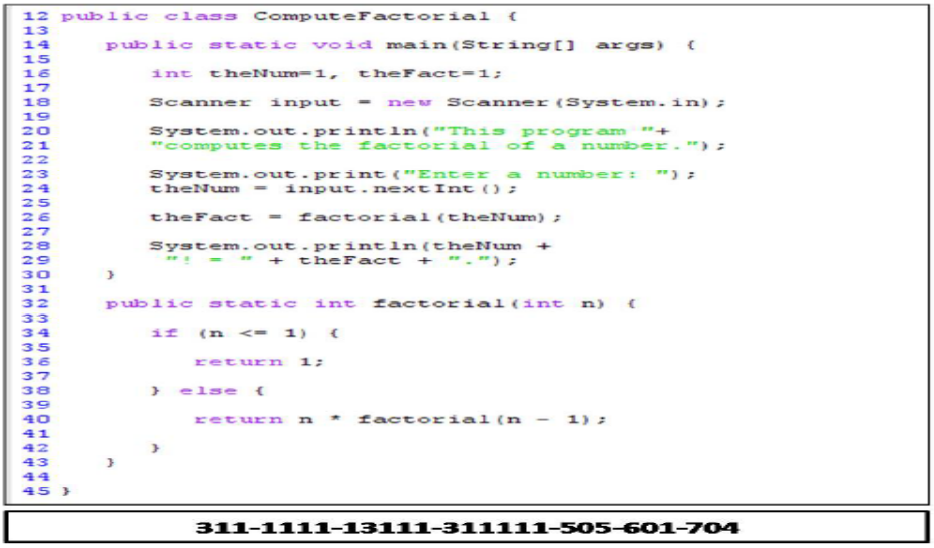
Como resultado, el patrón de identificación del primer modelo es el que mejor coincide con el patrón de identificación del estudiante. La calificación es asignada basándose en el programa del primer modelo.

2.3.3. Framework de eGrader

El framework de eGrader consta de 3 componentes : el calificador de código fuente, el generador de informes y el generador de sesiones de calificación. La pantalla principal de eGrader se muestra en la figura 2.14.

Generador de sesiones de calificación

EGrader soporta tanto la generación como el guardado de las sesiones de calificación. Generar una sesión de calificación es fácil, flexible y rápida. Una sesión de calificación es generada a través de 3 pasos : crear una lista de modelos, crear criterios de valoración y crear nueva sesión de calificación.



```
12 public class ComputeFactorial {
13
14     public static void main(String[] args) {
15
16         int theNum=1, theFact=1;
17
18         Scanner input = new Scanner(System.in);
19
20         System.out.println("This program "+
21         "computes the factorial of a number.");
22
23         System.out.print("Enter a number: ");
24         theNum = input.nextInt();
25
26         theFact = factorial(theNum);
27
28         System.out.println(theNum +
29           "! = " + theFact + ".");
30     }
31
32     public static int factorial(int n) {
33
34         if (n <= 1) {
35
36             return 1;
37
38         } else {
39
40             return n * factorial(n - 1);
41
42         }
43     }
44
45 }
```

311-1111-13111-311111-505-601-704

Figura 2.12: Solución de un estudiante

La creación de lista de modelos se hace simplemente añadiendo soluciones modelo, donde los patrones de identificación son generados automáticamente. Cada patrón de identificación representa la estructura de su solución modelo. Una vez que el patrón de identificación es generado, aparece una caja de diálogo mostrando el patrón de identificación y proporcionando la posibilidad de modificarlo.

Para crear y valorar un esquema, los criterios de valoración se dividen en 5 categorías :

1. Declaraciónes de condiciones.
2. Declaraciónes de bucles.
3. Llamadas a métodos recursivos y no recursivos.
4. Excepciones.
5. Variables, clases y llamadas a métodos de librerías.

La figura 2.15 muestra un frame de *Criterios de valoración*. Cada categoría proporciona campos de entrada para medir los pesos de las categorías y las penalizaciones por controles extras. Una categoría es añadida al proceso de calificación si tiene un peso mayor que 0. Si el valor de penalización de una categoría es mayor que 0, un estudiante que use controles extra (más de

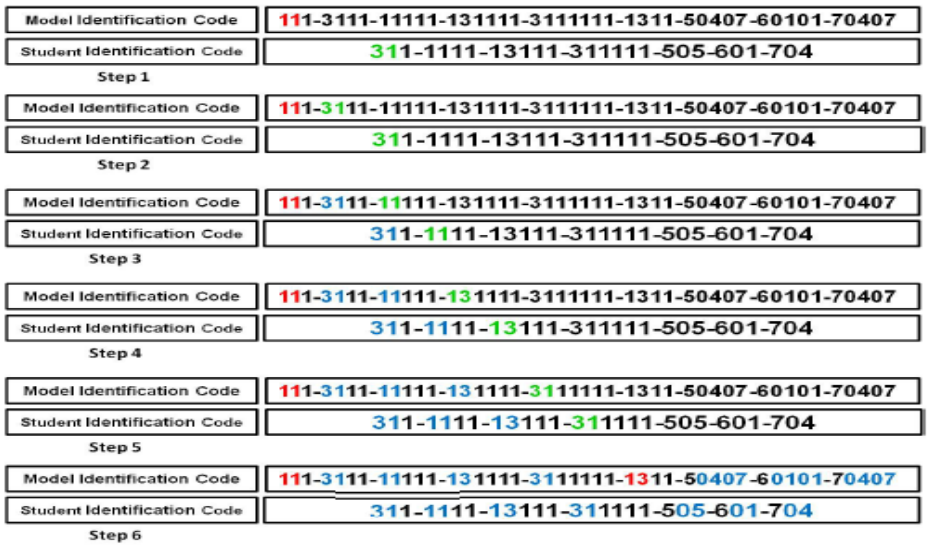


Figura 2.13: Proceso de comparación entre solución del alumno y solución del modelo

los que se requieren en el programa) de esa categoría será penalizado.

Una sesión de calificación es creada a través de un diálogo de Nueva Sesión de Calificación. En este diálogo se necesitan añadir 3 archivos : un archivo con el conjunto de soluciones, un archivo con los criterios de valoraciñ y un archivo de test de JUnit con una opción para especificar los pesos para el análisis dinámico. Una vez que la sesión de calificación para un problema es generada, el proceso de calificación puede tomar lugar.

Calificador de Código Fuente

Como la mayoría de los sistemas existentes hacen, el código fuente presentado necesita ser un archivo comprimido llamado con el número de identificación del usuario. Esta estrategia de entrega y de nombramiento se escoge para no cargar al profesor con la tarea de buscar los archivos requeridos en diferentes carpetas y realizar un seguimiento de que entregas pertenecen a que alumno. El proceso de calificación consta de los siguientes pasos:

1. Cargar sesión de calificación : La lista de soluciones, carpetas y patrones de identificación se cargan en una tabla formulario en el frame principal del eGrader.

2. Cagar los archivos comprimidos presentados especificando su carpeta

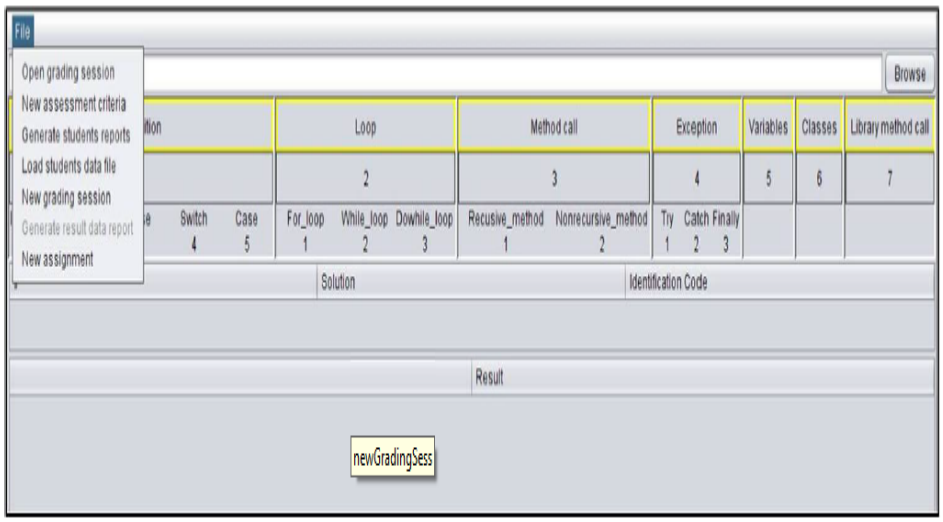


Figura 2.14: Ventana principal eGrader

3. Las entregas serán calificadas y su salida se insertará en otra tabla.

En esta etapa, el proceso de calificación esta completado. La lista de los nombres de los estudiantes junto con sus detalles está en un fichero Excel que es cargado en el eGrader.

El calificador empieza ejecutando el código fuente usando la clase JUnit (test dinámico). Si el test dinámico tiene éxito, el sistema continúa con el test estático. De otra manera, se genera un informe indicando que esta entrega necesita ser revisada por el profesor.

Generador de informes

EGrader no sólo califica código Java eficientemente, sino que también proporciona al profesor información detallada del proceso de calificación. Ayuda a analizar los conocimientos de los alumnos de los conceptos básicos de programación. Hay 2 tipos de informes que son producidos por eGrader : informes de clase e informes de valoración del estudiante. Empezaremos hablando de los informes de valoración del estudiante.

Después de que el proceso de calificación es completado y el archivo de datos del estudiante (un archivo excel que incluye los nombres de los estudiantes y los números de identificación) es cargado, se generan los informes de los estudiantes. Tales informes constan de 4 secciones :

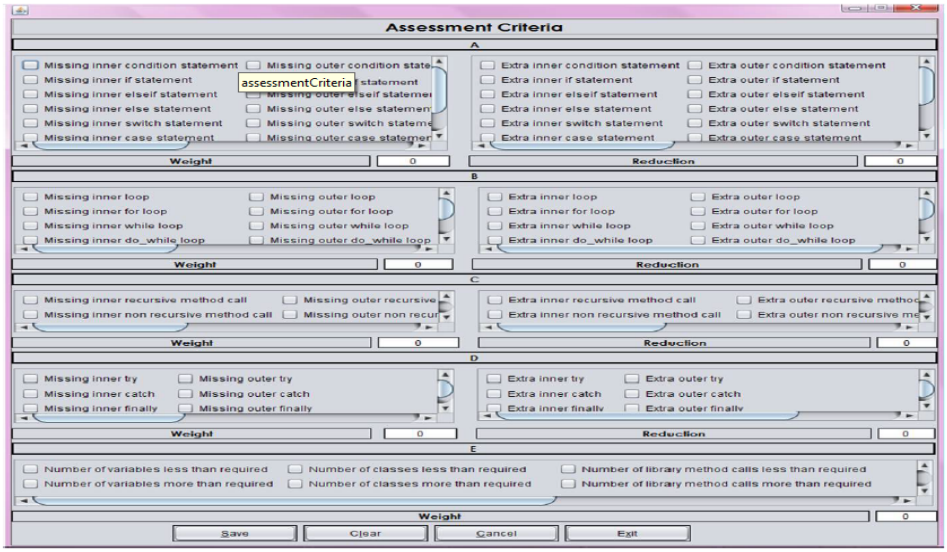


Figura 2.15: Criterios de valoración

- **Indentificación :** Contiene información del estudiante como el nombre, el número de identificación, y el resultado de puntuar su entrega.
- **Puntuación :** Muestra los detalles del esquema de puntuación después de aplicar tanto el test estático como el dinámico. El resultado del test dinámico incluye el número total de tests y el número de tests que fallarón. La parte estática muestra las 5 categorías generales y la nota para cada una, si se requiere. En el caso de encontrar errores, se insertará un mensaje para indicar la fuente del error. Las notas son deducidas basadas en el esquema original de puntuación establecido por el profesor.
- **Solución modelo :** Apunta a la solución modelo que mejor coincide con la entrega del estudiante.
- **Código original :** Muestra la solución del estudiante. Una coincidencia entre la estructura de la solución modelo y la estructura de la entrega del estudiante es mostrada usando coincidencia de colores entre las estructuras de control correspondientes.

Un informe para la entrega de un estudiante que contenga errores de sintaxis solo consiste en una parte, la cuál indica que la entrega tiene errores y que tiene que ser comprobada por un profesor. La nota total para esta entrega es 0.

Por último, vamos a hablar del informe de clase.

Un informe de clase es un informe resúmen del rendimiento de la clase entera en una entrega especifica. Este informe consta de 3 partes (3 hojas excel) las cuáles son : estadísticas, detalles del test dinámico y detalles del test estático.

Información útil tal como el nivel de dificultad de la entrega, el número de estudiantes que lograrón presentar una solución y las soluciones más y menos comunes, son resumidas en la parte de las estadísticas. La parte de las estadísticas consta de los siguientes datos :

- Número de entregas de los estudiantes para una entrega basada en el número de entregas calificadas.
- Número de soluciones modelo.
- Modelo solución más popular.
- Modelo solución menos popular.
- Número de tests de unidad usados para testear cada entrega.
- Número de entregas que fallarón todos los tests. Este número indica las entregas que fallarón todos los tests en la clase de test JUnit.
- Número de entregas fallidas debido a errores de sintaxis

La parte de detalles del test dinámico proporciona una visión general del rendimiento de la clase. Muestra los siguientes datos:

- Tests fallidas al ejecutar por la entrega de un estudiante junto al número de estudiantes que fallarón cada test.
- Lista de errores en tiempo de ejecución.
- Otras estadísticas útiles tales como nota promedio, mínima y máxima.

La parte de detalles del test estático proporciona información del rendimiento de la clase en las 5 categorías generales. Esta parte consta de los siguientes datos:

- Requerimientos de entrega el cuál contiene 5 categorías, donde cada una tiene 3 medidas : nota promedio, la nota más alta y la más baja, si se requiere la categoría. De otra manera, la categoría serán informadas como no requeridas.
- Otras estadísticas útiles tales como las notas promedios, mínimas y máximas.

2.3.4. Resultados experimentales

EGrader ha sido evaluado por un conjunto representativo de datos de soluciones de estudiantes en cursos de introducción a la programación en Java en la Universidad de Sharjah. Este conjunto de datos consta de entregas de estudiantes para 2 semestres con un total de 191 entregas con un promedio de 24 estudiantes por clase. El conjunto de entregas cubre 9 problemas diferentes.

Se usarón 4 tipos de entregas de programación las cuáles eran :

- **Entrega 1 :** Prueba la habilidad para usar variables, declaraciones de entradas, expresiones Java, cálculos matemáticos y declaraciones de salidas.
- **Entrega 2 :** Prueba la habilidad para usar estructuras de control de condiciones tales como if/else-if/else y declaraciones switch
- **Entrega 3 :** Prueba la habilidad para usar métodos recursivos y no recursivos.
- **Entrega 4 :** Prueba la habilidad para usar arrays.

Estamos usando 4 medidas de rendimiento para evaluar el rendimiento de eGrader :

- **Sensibilidad :** Mide cuantas entregas correctas son de hecho premiadas.
- **Especificidad :** Mide cuantas entregas erróneas son penalizadas.
- **Precisión :** Mide cuantas entregas premiadas son correctas.
- **Exactitud :** Mide el número de entregas correctamente clasificadas.

La evaluación muestra una alta tasa de éxito representada por la medidas de rendimiento descritas anteriormente. Los resultados aparecen en la figura 2.16.

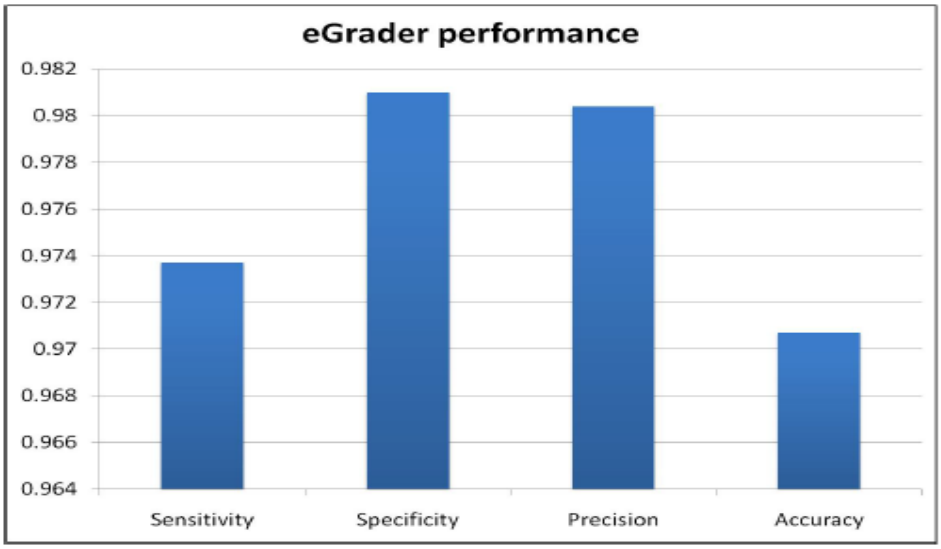


Figura 2.16: Evaluación del rendimiento de eGrader

Capítulo 3

EagerBT

Resumen: En este capítulo hablaremos de la primera parte del proyecto encargada del Framework EagerBT. En el apartado 3.1 describiremos que es el Framework EagerBT y hablaremos de algunos ejemplos de lo que se puede hacer con este framework sin profundizar en cómo se han creado. En la sección 3.2 hablaremos acerca de los elementos necesarios para la creación y ejecución de los árboles de comportamiento (nodos, contexto, Log, Observer y el ejecutor de árboles) En la sección 3.1 trataremos más en profundidad cada uno de estos elementos hablando de la manera en la que se han implementado en este framework. Por último, en la sección 3.4 hablaremos de algunos ejemplos de nodos complejos que se han creado a partir de nodos básicos.

3.1. Introducción

EagerBT es un framework para árboles de comportamiento que permite construirlos desde código y ejecutarlos.

Antes de entrar en profundidad a explicar cada uno de los elementos de los que está formado el framework, vamos a ver varios ejemplos de los árboles de comportamiento que se pueden crear. Dichas explicaciones vienen acompañadas con una figura que representa la estructura del árbol y con una interfaz gráfica simple para ver el resultado de la ejecución del árbol. Además, el primer ejemplo también viene con el código de creación del árbol (para el resto de ejemplos el árbol se construiría de forma parecida).

3.1.1. Ejemplo 1

Explicación

En este primer ejemplo, vamos a mostrar el funcionamiento del nodo *Switch*. La funcionalidad de este nodo es la de ejecutar uno de los nodos hijo de lo que está formado.

Código

```
 private static Composite createInstruction2Aux(){
   return new Sequential().addNode(new PrintAction(new Constant
                                ("Instruccion 2.1")))
                          .addNode(new PrintAction(new Constant
                           ("Instruccion 2.2")));
}

private static Composite createTest2(){
   return new Switch()
            .addNode(new PrintAction(new Constant("Instruccion 0")))
           .addNode(new PrintAction(new Constant("Instruccion 1")))
           .addNode(createInstruction2Aux())
           .addNode(new PrintAction(new Constant("Instruccion 3")))
           .addNode(new PrintAction(new Constant("Instruccion 4")));
}
```

Árbol de comportamiento

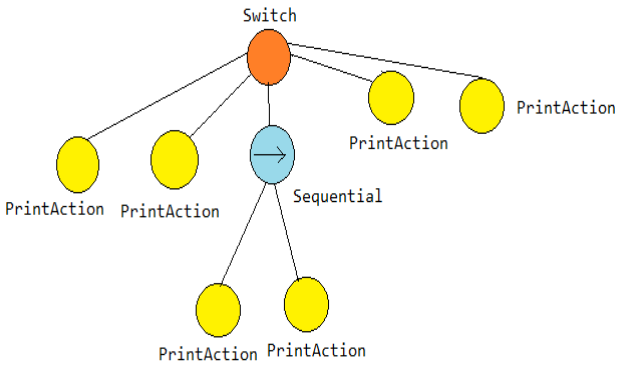


Figura 3.1: Árbol de comportamiento del ejemplo 1

Explicación del árbol de comportamiento

De izquierda a derecha los nodos *PrintAction* muestran los siguientes

mensajes:

- Instruccion 0
- Instruccion 1
- Instruccion 2.1
- Instruccion 2.1
- Instruccion 3
- Instruccion 4

En el caso de que la variable *numInsSwitch* tenga el valor 2, se ejecuta el nodo *Sequential* y entonces se muestran 2 mensajes.

Resultado de la ejecución del árbol de comportamiento

Supongamos que *numInsSwitch* tiene el valor 2. La ejecución del árbol de comportamiento es la siguiente:

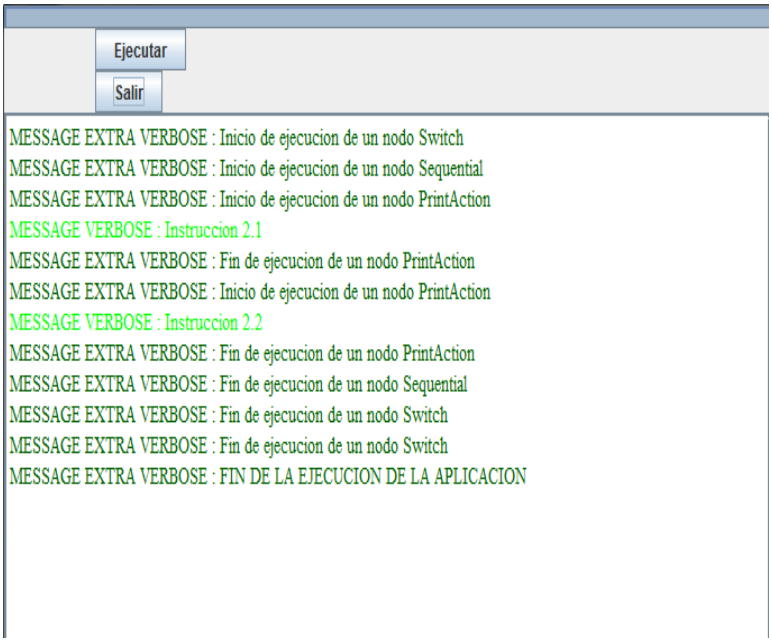


Figura 3.2: Ejecución del ejemplo 1

Como podemos observar , efectivamente, se muestran 2 mensajes por pantalla : *Instruccion 2.1* y *Instruccion 2.2* que son los que se tenían que mostrar.

3.1.2. Ejemplo 2

Explicación

En esta ejemplo, vamos a mostrar el funcionamiento del nodo *If*. La funcionalidad de este nodo es ejecutar un nodo condición y dependiendo del resultado de una comparación ejecutar un nodo (la parte del *Then*) u otro (la parte del *Else*).

Árbol de comportamiento

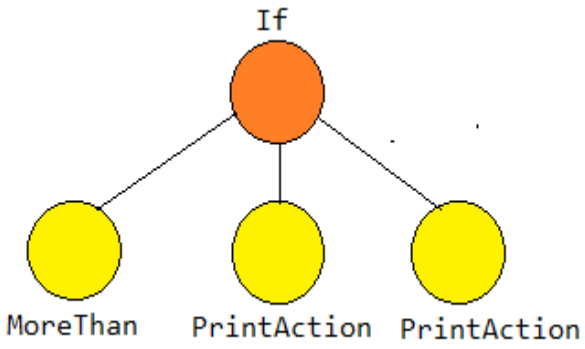


Figura 3.3: Árbol de comportamiento del ejemplo 2

Explicación del árbol de comportamiento

El proceso de ejecución del árbol de comportamiento es el siguiente:

- Se ejecuta el nodo condición. En este caso es un nodo *MoreThan* que comprueba si un valor es mayor que otro. En este ejemplo, compara si un valor constante (5) es mayor que el valor de la variable *testkey2* almacenada en el contexto (en este ejemplo tiene valor 10).
- Si la comparación devuelve *true* se muestra el mensaje *Se ejecuta la rama Then*.
- Si la comparación devuelve *false* se muestra el mensaje *Se ejecuta la rama Else*.

En nuestro ejemplo, como 5 no es mayor que 10, se tendría que mostrar el mensaje *Se ejecuta la rama Else*.

Resultado de la ejecución del árbol de comportamiento

El resultado de la ejecución es el que se muestra en la figura 3.4.

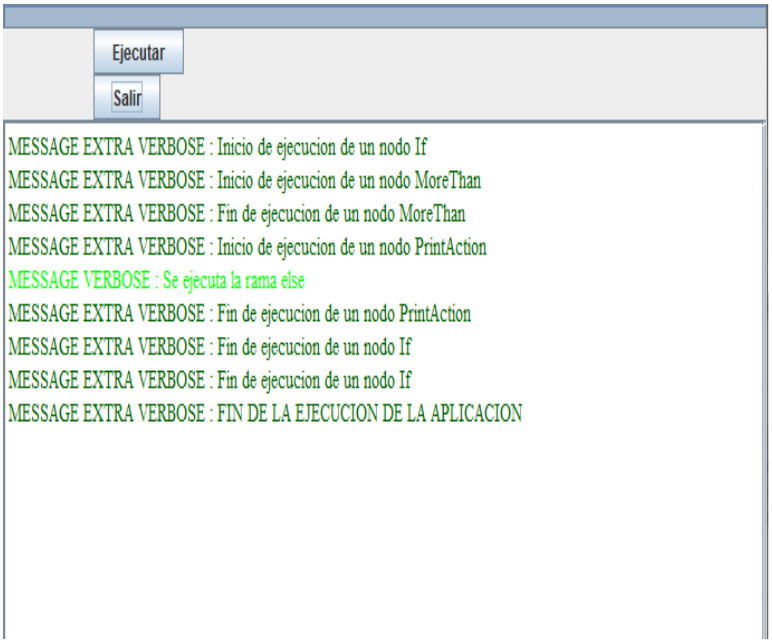


Figura 3.4: Ejecución del ejemplo 2

Como podemos observar , efectivamente, se muestra el mensaje *Se ejecuta la rama else*.

3.1.3. Ejemplo 3

Explicación

En este ejemplo, vamos a mostrar el funcionamiento del nodo *Random*. la funcionalidad de este nodo es ejecutar un nodo aleatorio de la lista de nodos hijo.

Árbol de comportamiento

El árbol de comportamiento es de la Figura 3.5.

Explicación del árbol de comportamiento

De izquierda a derecha los nodos *PrintAction* muestran los siguientes menajes:

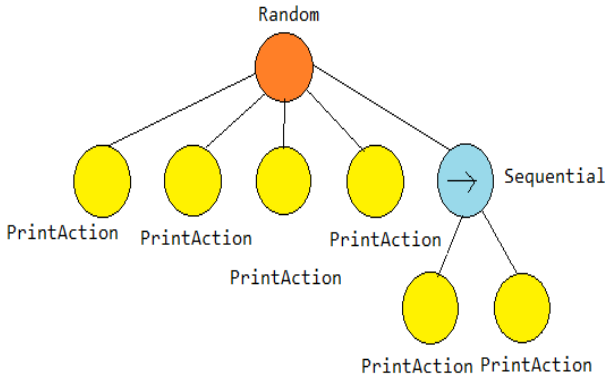


Figura 3.5: Árbol de comportamiento del ejemplo 3

- Instruccion 0
- Instruccion 1
- Instruccion 2
- Instruccion 3
- Instruccion 4.1
- Instruccion 4.2

Resultado de la ejecución del árbol de comportamiento

Como el nodo de las lista de nodos hijo es aleatorio, vamos a ejecutar 2 veces el árbol de comportamiento (figuras 3.6 y 3.7)

Como podemos observar en la figura 3.6, se ha ejecutado el nodo *Sequential* (nodo que ejecuta todo sus hijos en orden).

Como podemos observar en la figura 3.7, se ha ejecutado el primero nodo *PrintAction* y se muestra el mensaje *Instruccion 0*.

3.1.4. Ejemplo 4

Explicación

En este ejemplo, vamos a comprobar el perfecto funcionamiento del nodo *For*. La funcionalidad de este nodo es ejecutar una lista de nodos hijo un número de veces igual al valor de la variable *numIter* almacenada en el contexto.

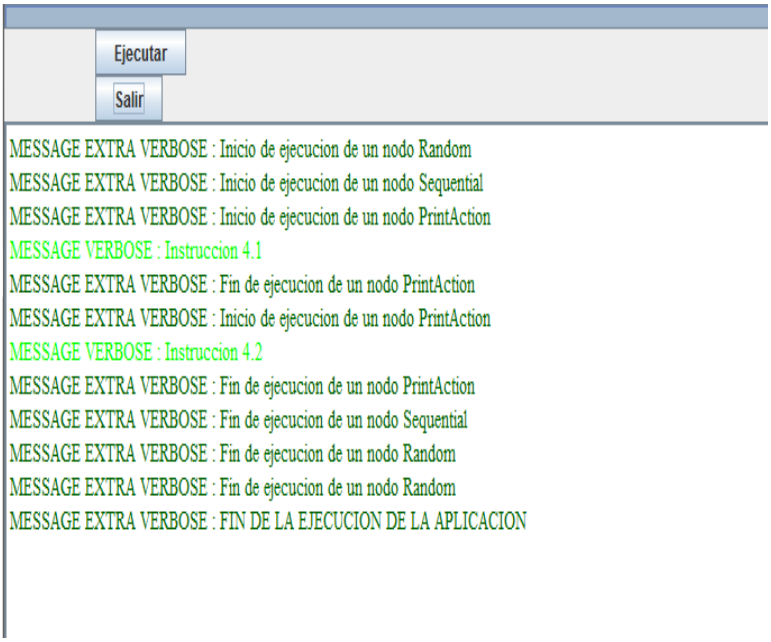


Figura 3.6: Ejecución del ejemplo 3

Árbol de comportamiento

El árbol de comportamiento es de la Figura 3.8.

Explicación del árbol de comportamiento

El proceso de ejecución del árbol de comportamiento es el siguiente:

- Se ejecuta el nodo For. En este ejemplo, en cada iteración (en total son 10 iteraciones ya que en este ejemplo la variable *numIter* almacenada en el contexto tiene valor 10),se ejecutan 2 nodos *IncrementAction* que aumentan cada uno el valor de la variable *key5* almacenada en el contexto sumándole a ese valor 5.
- El nodo *ReadAction* muestra el valor final de esa variable.

En nuestro ejemplo, el valor de la variable *key5* aumenta su valor 10 unidades (ya que se le suma 5 2 veces) 10 veces (1 por cada iteración). Por lo tanto, el valor final de la variable es 100 que es el que se va a mostrar cuando ejecutemos el árbol.

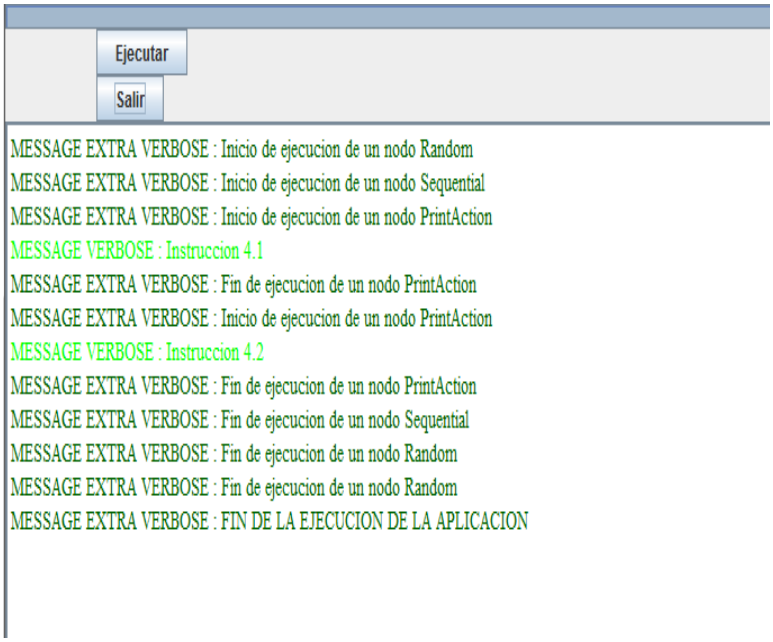


Figura 3.7: Ejecución del ejemplo 3

Resultado de la ejecución del árbol de comportamiento

El resultado de ejecución del árbol de comportamiento es de la Figura 3.9.
En la figura no se muestra toda la ejecución ya que en la parte donde se van mostrando los mensajes es demasiado pequeña como para que quepan todos y por eso aparece una barra de desplazamiento vertical.
Como podemos observar , efectivamente, se muestra 100 como el valor final de la variable.

3.2. Explicación

El primer paso para crear un validador de prácticas es construir los árboles de comportamiento como explicamos en el primer capítulo.

3.2.1. Nodo

Recordemos que los árboles de comportamiento son estructuras que se crean con la finalidad de organizar el comportamiento de nuestro sistema y permitir trabajar con él de forma sencilla. Para la creación de un árbol de comportamiento podemos usar diferentes tipos de nodos los cuáles tendrán

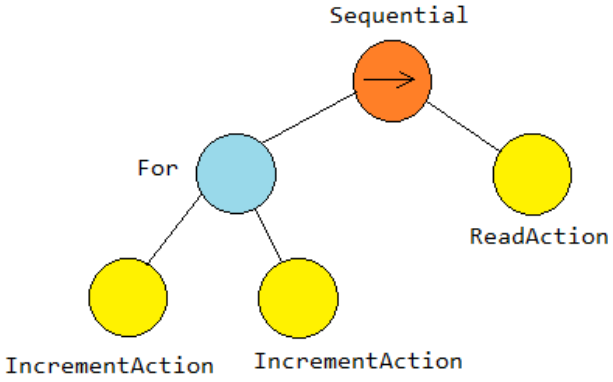


Figura 3.8: Árbol de comportamiento del ejemplo 4

cada uno una funcionalidad diferente.

Los nodos creados en este framework se pueden dividir en 2 categorÃas :

- **Nodos Internos :** Son los nodos encargados de la implementación de estructuras de control (if, while, for,..).
- **Nodos Hoja :** Son los nodos encargados de realizar las acciones importantes.

En todos los árboles de comportamiento que se van a crear, el nodo raíz (el primer nodo que se ejecuta del árbol) va a ser siempre un nodo secuencial nodo con una lista de hijos que se van ejecutando uno a uno hasta que se ejecuten correctamente todos o falle alguno).

En esta primera parte se han creado varios tipos de nodos : nodos para la implementación de condiciones, nodos para el manejo del contexto, nodos encargados de la implementación de las estructuras de control y , por último, nodos que realizan otras tareas simples. De los nodos creados para cada uno de estos tipos se hablará en la siguiente sección.

3.2.2. Contexto

Un contexto es como una especie de pizarra donde vamos guardando variables junto a sus valores (1 solo valor o una lista de valores). Va a ser en el contexto donde los nodos van a escribir valores para que otros lo lean. A veces puede darse el caso en el que un nodo escriba el valor de la variable A, a continuación otro nodo sobrescriba el valor de la variable A y que después otro nodo quiera leer el primer valor que se le asignó a la variable. Para que

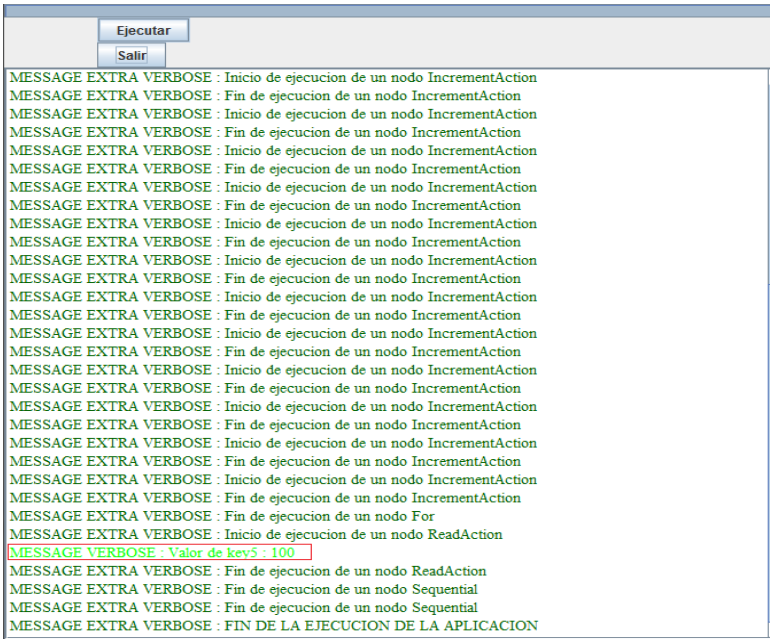


Figura 3.9: Ejecución del ejemplo 4

esto sea posible el contexto en vez de ser una tabla será una pila de tablas. De esta manera, el segundo nodo apilará un nuevo contexto, escribirá en él el valor de la variable, hará con él lo que tenga que hacer y desapilará el contexto de la cima de la pila (ese mismo nodo u otro diferente) por lo que el tercer nodo leerá el valor correcto de la variable.

Las posibles operaciones que se pueden realizar sobre el contexto son las siguientes:

- Añadir una tabla vacía en la pila.
- Eliminar la tabla de la cima de la pila.
- Anádir un elemento (variable y valor) en la tabla de la cima de la pila.
- Buscar el valor de una variable en la tabla de la cima y si no se encuentra, se busca en las que tiene debajo.

A continuación veamos un ejemplo de un posible contexto en el que ya se han escrito 5 variables con sus respectivos valores :

3.2.3. BTExecutor (Ejecutor de árboles de comportamiento)

Por último y lo más importante junto a los nodos, es el BTExecutor el cuál es el encargado de ejecutar los nodos que forman el árbol de comportamiento. Los nodos se ejecutan a través de este pasándole también el

Nombre	Valor
Numero1	12
Numero2	13
Resultado Suma	25
Resultado de la condicion	false
Numero de iteraciones	20

Tabla 3.1: Ejemplo de contexto

contexto. Además es el encargado de ir avisando de la traza de ejecución del árbol de comportamiento (aumentar el número de nodos ejecutados, avisar del tipo de nodo que se empieza a ejecutar y del tipo de nodo que acaba su ejecución).

3.2.4. Supervisión externa del proceso

Para llevar a cabo una supervisión externa del proceso usamos el patrón `Observer`. Para quien no lo sepa, con el patrón Observer se define una dependencia entre uno a muchos, de tal forma que cuando que cuando el objeto cambie de estado, todos sus objetos dependientes serán notificados automáticamente.

En nuestra aplicación y frameworks, los objetos dependientes son las interfaces gráficas que informarán al usuario de la aplicación de la evolución del proceso de ejecución del árbol de comportamiento; y el objeto que cambia de estado es un objeto de la clase BTExecutor.

Además, nuestra aplicación y framework admite varios tipos de Observadores los cuáles se explicarÃ¡n en las correspondientes secciones del documento.

3.2.5. Tipos de nodos

Los distintos nodos que se han creado en este framework son los siguientes:

- Nodos para comparar 2 valores (<,<=,>,>=,=)
- Nodos que devuelven siempre COMPLETED o FAILED.
- Nodos para manejar el contexto : añadir tablas, eliminar tablas y mover variables de una tabla a otra.
- Nodos para representar estructuras de control : if, while, doWhile, for, finally entre otros.
- Nodos para realizar tareas simples como mostrar mensajes, mostrar el valor de una variable del contexto, terminar una ejecución con éxito

o error automáticamente o almacenar una variable y un valor en el contexto.

3.2.6. Parámetros de acciones y de nodos

Hay veces en que necesitamos el valor de una variable la cuál puede ser tanto una variable almacenada en el contexto como un valor constante. Por ello, se ha creado 2 tipos de datos auxiliares : Constant (el valor es una constante) y Param (el valor se obtiene del contexto). Así que, cuando haya que definir dentro de una clase algunas variables, usaremos estos tipos para decirle al programa de donde tiene que obtener el valor de dicha variable.

Para ver la diferencia entre ambos tipos, vamos a estudiar la ejecución de un nodo `PrintAction` creado y ejecutado mediante el siguiente código:

```
public int main(){
    Node tree = new PrintAction(new Constant
            (''Esto es un mensaje constante''));
    Node tree2 = new PrintAction(new Param(''mensaje''));
    bte.executeTree(ctx,tree);
    bte.executeTree(ctx,tree2);
}
```

Suponemos que ya se creó el contexto `ctx` y el BTExecutor `bte`, y que se ha preparado todo lo relacionado con el `Log` y el `Observer`. También suponemos que el contexto tiene una variable `mensaje` con valor "Esto es un mensaje sacado del contexto".

Como el árbol `tree` se definió pasándole como parámetro una variable del tipo `Constant` con valor "Esto es un mensaje constante", cuando se ejecute el árbol será éste el mensaje que se muestre.

Como el árbol `tree2` se definió pasándole como parámetro una variable del tipo `Param` con valor "mensaje, cuando se ejecute el árbol, se mostrará el valor de la variable "mensaje" almacenada en el contexto (en este caso se muestra "Esto es un mensaje sacado del contexto").

3.2.7. Result

Tipo de dato usado para representar si un nodo acaba su ejecución con éxito (COMPLETED) o falla (FAILED).

3.2.8. Log

Durante el proceso de ejecución de un árbol de comportamiento, necesitamos tener un mecanismo que informe de la evolución de dicho proceso de ejecución y de los posibles mensajes que genere dicho proceso.

Para llevar a cabo todo esto, se ha definido el concepto de `Log`, el cuál es una especie de "informe" por donde se muestran los mensajes que se generan durante el proceso de ejecución.

Ejemplos de nodos que usan el `Log` son los siguientes:

- **PrintAction** : Muestra un mensaje el cuál se le puede pasar como parámetro al nodo, o también puede cogerlo del contexto.
- **ReadAction** : Muestra el nombre de una variable y el valor que contiene en el contexto.
- **ShowFinalValue** : Muestra el valor final de la ejecución de un árbol de comportamiento.

3.3. Implementación

En esta sección se tratará la implementación en Java de cada uno de los elementos descritos en el apartado anterior.

3.3.1. Contexto

Como se comentó anteriormente, el contexto se implementa mediante una pila de Tablas Hash donde la clave es de tipo String y el valor es del tipo Object para admitir cualquier tipo de valor para las variables.

Para ver lo que hace cada una de las operaciones que se permiten realizar sobre el contexto, se van a mostrar en las figuras ejemplos de las pilas de tablas que forman el contexto. Para simplificar las figuras, aparecen las siguientes abreviaturas:

- **N.V** : Nombre de la variable.
- **V.V** : Valor de la variable.
- **Ni** : Nombre de la variable "i".
- **Vi** : Valor de la variable "i".

Los principales métodos de la clase `Contexto.java`, la cuál es la clase que lo implementa, son los siguientes:

- **Context()** : Constructora la cual inicializa la pila e inserta una tabla hash vacía.

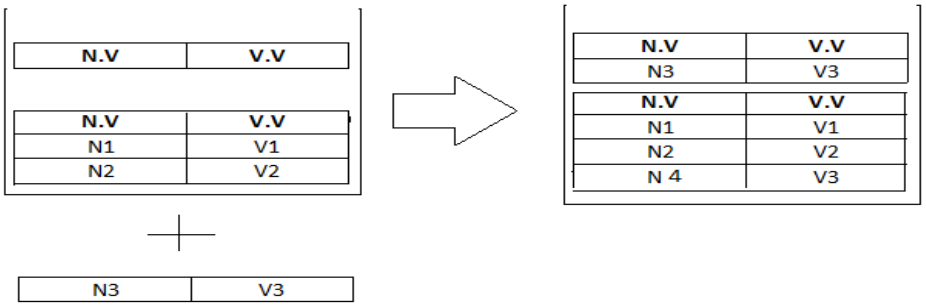


Figura 3.10: Insertar elemento

- **addElement(String clave,Object valor)** : Añade la variable "clave" con valor "valor" en la tabla de la cima de la pila. Ver figura 3.10.
- **pushTable()** : Añade una nueva tabla vacía en la pila.

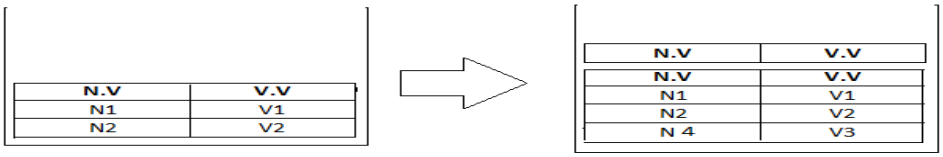


Figura 3.11: Añadir tabla

- **popTable()** : Elimina la tabla de la cima de la pila.

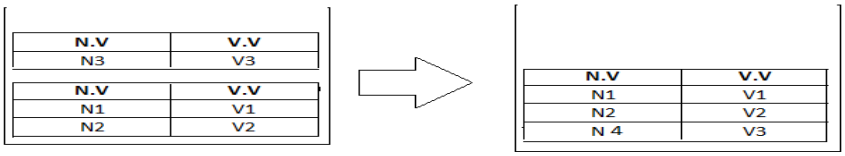


Figura 3.12: Eliminar tabla

- **getValue(String clave)** : Consulta en la tabla de la cima de la pila el valor de la variable con nombre "clave" y devuelve dicho valor. Si la variable no está en la tabla de la cima de la pila, mira en las siguientes tablas hasta encontrar la variable.

3.3.2. Tipos auxiliares : Param, Constant y Value

Como hemos mencionado anteriormente, algunos nodos necesitan para su creación o para su ejecución el valor de una variable, el cuál puede ser o una constante o el valor que se obtiene de consultarlo en el contexto que se

le pasa al nodo. Para que no sea el nodo el que tenga que averiguar de donde sacar el valor se definen las siguientes clases:

- **Constant** : El valor es constante.
- **Param** : El valor se obtiene de un contexto.

`Value` es la interfaz que implementan tanto la clase `Param` como la clase `Constant`.

La clase `Constant` tiene un campo `constantValue` de tipo Object el cuál representa el valor que va a tener el objeto que se defina de esta clase.
La clase `Param` tiene un campo `nameParam` de tipo String el cuál representa el nombre de la variable que se va a usar para consultar en el contexto el valor del objeto que se defina de esta clase.

Ambas clases implementan los siguientes métodos:

- **getValue()** : Devuelve el valor del objeto que se defina de esta clase. En el caso de la clase `Constant` se devuelve directamente el valor del campo `constantValue`. En el caso de la clase Param, se consulta en el contexto el valor de la variable `nameParam` y se devuelve dicho valor.
- **setValue(ctx,value)** : En el caso de la clase `Constant` establece el valor de `constantValue` a `value`. En el caso de la clase `Param` se añade al contexto como clave `nameParam` y como valor `value`.

Además, cada clase tiene una constructora para asignarle valores a los campos de las clases.

3.3.3. Result

Tipo enumerado formado por 2 valores : COMPLETED y FAILED. Cuando se ejecute una tarea o nodo se devolverá COMPLETED si la ejecución tiene éxito y FAILED si falla la ejecución.

3.3.4. Nodos

Los nodos son los elementos más básicos que se usan a la hora de construir un árbol de comportamiento.

Un nodo puede tener varios hijos, por lo que un nodo puede verse como si fuera un árbol.

La acción depende del tipo de nodo (método `execute`) y la forma/orden en que se ejecutan los hijos también.

La interfaz `Node` es la encargada de la implementación de los nodos. Dicha interfaz contiene los siguientes métodos:

- **getNumNodes()** : Devuelve el número de nodos de los que consta dicho nodo. Esto puede resultar un poco extraño pero existen algunos nodos compuestos como pueden ser los nodos que implementan las estructuras de control. Este método devuelve 1 más el número de nodos de los que consta (hay nodos que tienen como campo otros nodos).
- **toString()** : Devuelve el nombre del nodo.
- **execute(bte,ctx)** : Ejecuta dicho nodo usando como BTExecutor `bte` y como contexto `ctx`.

  Para ejecutar un hijo se llama al método `execute` del `bte` pasándole como parámetros el contexto `ctx` y el nodo hijo.

  Algunas veces, avisa al `bte` que se va a repetir la ejecución de un nodo hijo o si se va a saltar uno (cuando algún nodo hijo falle e impida que el resto de hijos se ejecuten).

  Devuelve COMPLETED si la ejecución tiene éxito y FAILED en caso contrario.

Los detalles de implementación de cada método dependerán del tipo de nodo.

3.3.5. BTExecutor

Es la clase encargada de controlar la ejecución de un árbol. El usuario le pide la ejecución y éste delega en el método `execute()` del nodo. Tiene como único atributo una lista de objetos de la clase Observer.
Esta clase contiene los siguientes métodos:

- **executeTree(tree,ctx)** : Se encarga de ejecutar el árbol que se le pasa por parámetro usando el contexto que también se le pasa por parámetro. Para ello se realizan los siguientes pasos:

  1. Informa a los observadores de la lista que se va a empezar a ejecutar el árbol de comportamiento.
  2. Se lanza la ejecución del nodo raíz del árbol de comportamiento pasándole el contexto y el propio BTExecutor.
  3. Informa a los observadores de la lista que ha acabado la ejecución del árbol de comportamiento.

- **addObserver(obs)** : Añade el objeto de la clase Observer que se le pasa por parámetro a la lista de observadores.
- **skipNodes(numNodes)** : Método que es llamado por los nodos durante su ejecución para informar de que va a dejar de ejecutar (saltarse) algunos de sus hijos. `NumNodes` es el número de nodos que forman los nodos hijo que se saltan.
- **repeatNodes(numNodes)** : Método que es llamado por los nodos durante su ejecución para informar de que va a repetir la ejecución de algunos de sus hijos. `NumNodes` es el número de nodos que forman los nodos hijo que se repiten.
- **executeNode(node,ctx)** : Se encarga de ejecutar el nodo que se le pasa por parámetro usando el contexto que también se le pasa por parámetro. Para ello se realizan los siguientes pasos:
  1. Informa a los observadores de la lista que se va a empezar a ejecutar un nodo.
  2. Se lanza la ejecución del nodo pasándole el contexto y el propio BTExecutor.
  3. Informa a los observadores de la lista que ha acabado la ejecución del nodo.

3.3.6. Log

Es una clase estática usada por los árboles de comportamiento para escribir mensajes.

Permite 2 tipos de mensajes : mensajes de información y mensajes importantes. Consta de 2 métodos encargados de mostrar dichos mensajes a través de una salida que se determina mediante un atributo de la interfaz Output.

En esta primera parte, solo tenemos un tipo de salida : la salida estándar (`System.out`) la cual es implementada mediante la clase `ConsoleOutput`.

3.3.7. Observer

Interfaz que se encarga de la aplicación del patrón Observer en nuestra aplicación y framework. Para quien no lo sepa, con el patrón Observer se define una dependencia entre uno a muchos, de tal forma que cuando que cuando el objeto cambie de estado, todos sus objetos dependientes serán notificados automáticamente.

En nuestra aplicación y framework, los objetos dependientes son las interfaces gráficas que informarán al usuario de la aplicación de la evolución del proceso de ejecución del árbol de comportamiento; y el objeto que cambia de estado es un objeto de la clase BTExecutor.
Las diferentes notificaciones a los objetos dependientes son las siguientes:

- Se inicia la ejecución del árbol de comportamiento : se realiza mediante el método `onStartTreeExecution(tree,ctx)` de la interfaz `Observer` donde `tree` es el árbol que se empieza a ejecutar y `ctx` el contexto que se usa.

- Se acaba la ejecución del árbol de comportamiento : se realiza mediante el método `onEndTreeExecution(tree,ctx)` de la interfaz `Observer` donde `tree` es el árbol que ha terminado su ejecución y `ctx` el contexto que se usa.

- Se inicia la ejecución de un nodo : se realiza mediante el método `onStartNodeExecution(node)` de la interfaz `Observer` donde `node` es el ñode que se empieza a ejecutar.

- Se acaba la ejecución de un nodo : se realiza mediante el método `onEndNodeExecution(node)` de la interfaz `Observer` donde `node` es el ñode que se ha terminado de ejecutar.

- Se avanza un nodo en el árbol : se realiza mediante el método `advanceNode()` de la interfaz `Observer`.

- Se saltan varios nodos del árbol : se realiza mediante el método `skipNodes(numNodes)` de la interfaz `Observer` donde `numNodes` es el número de nodos que se saltan.

- Se repiten varios nodos del árbol : se realiza mediante el método `repeatNodes(numNodes)` de la interfaz `Observer` donde `numNodes` es el número de nodos que se repiten.

Para esta parte, se han implementado 2 `Observer`:

- **Consola** : Las notificaciones se muestran a través de la cónsola.

- **ProgressWindow** : Las notificaciones se muestran a través de una interfaz gráfica.

La interfaz gráfica a la que se ha hecho referencia se va a explicara continuación.

3.3.8. Interfaz gráfica : versión 1

Esta primera versión de la interfaz es la que ha usado para comprobar el perfecto funcionamiento de los nodos (y de los árboles derivados de éstos) creados para este framework.

Esta primera versión de la interfaz es demasiado simple, pero lo suficientemente útil como para ver el desarrollo de la ejecución de un árbol de comportamiento.

La estructura de la interfaz es la siguiente :

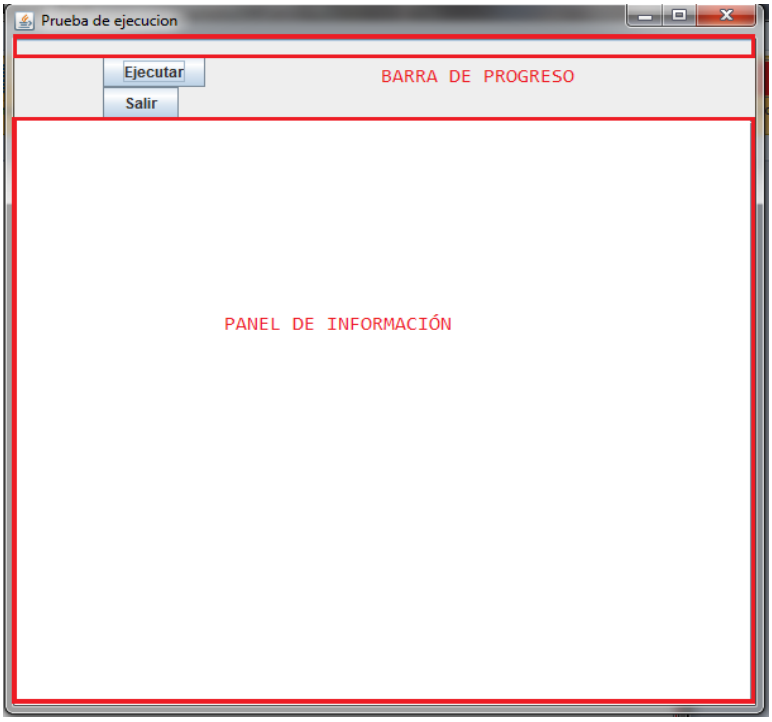


Figura 3.13: Primera versión de la interfaz gráfica

Los elementos más importantes de la interfaz son los siguientes:

- **Botón Salir** : Para salir de la aplicación.
- **Botón Ejecutar** : Para ejecutar el árbol de comportamiento correspondiente.
- **Panel de Información** : Para mostrar los mensajes de información que producen los nodos.
- **Barra de progreso** : Para ir mostrando el tanto por ciento de número de nodos ejecutados.

Los aspectos más importantes de la implementación de la interfaz gráfica son los siguientes:

- A la constructora de la interfaz se le pasa tanto el árbol de comportamiento que se quiere ejecutar, como el contexto y el BTExecutor que se van a usar.
- A dicho BTExecutor se le añade a la lista de observadores esta interfaz (la cuál implementa la interfaz gráfica) para que éste le pueda decir a la interfaz los mensajes que tiene que mostrar (nodos que inician y finalizan su ejecución) y cuando tiene que avanzar o retroceder la barra de progreso.
- La barra de progreso se crea con valor mínimo 0 y con valor máximo el número total de nodos del árbol de comportamiento que se va a ejecutar.
- El panel donde se van mostrando todos los mensajes de información acerca de la ejecución del árbol de comportamiento es del tipo `TextAreaOutput` para que vayan apareciendo los mensajes que van escribiendo cada nodo del árbol. Para ello, se establece el tipo de salida del Log como un objeto del tipo `TextAreaOutput` (clase que usa un objeto del tipo `JEditorPane` para mostrar los mensajes).
- Cuando se pulsa el botón `Ejecutar`, se crea un nuevo thread para ir mostrando los mensajes de información e ir avanzando la barra de progreso a la vez que se va ejecutando el árbol de comportamiento.

3.3.9. Tipos de nodos

A continuación describimos los nodos básicos que hemos implementado para el framework (no explicaremos el método `toString()` ya que lo único que hace es devolver el nombre del nodo y no requiere más explicaciones y tampoco el método `getNumNodes()` ya que lo único que hace es devolver la suma de nodos que forman ese nodo más 1):

- **Nodos de condiciones** : Su función es la de comparar 2 valores y almacenar el resultado de la comparación en el contexto usando una variable `condResult`. Cada nodo tiene 2 campos del tipo Value que son los valores a comparar. Los diferentes nodos de condición que se han implementado son los siguientes:
  - **Equal** : Comprueba si los valores de los 2 campos son iguales.
  - **False** : Siempre guarda como resultado False.
  - **True** : Siempre guarda como resultado True.

- **LessOrEqualThan** : Comprueba si el valor del campo 1 es menor o igual que el valor del campo 2.
- **LessThan** : Comprueba si el valor del campo 1 es menor que el valor del campo 2.
- **MoreOrEqualThan** : Comprueba si el valor del campo 1 es mayor o igual que el valor del campo 2.
- **MoreThan** : Comprueba si el valor del campo 1 es mayor que el valor del campo 2.

Como la ejecución de todos estos nodos no falla nunca, la ejecución siempre devuelve COMPLETED.

- **Nodos de manejo del contexto** : Se encargan de la modificación del contexto. Los nodos que pertenecen a esta categoría son los siguientes:

  - **PushContext** : Este nodo es el que se encarga de resolver el problema que mencionamos casi al principio del capítulo (un nodo escribe el valor de una variable en la tabla, un nodo sobrescribe ese valor pero después otro nodo quiere leer el valor que escribió el nodo del principio). Este nodo se encarga de insertar una nueva tabla vacía en la pila del contexto, ejecutar un nodo hijo (el que sobrescribe el valor) y cuando el nodo hijo se ha terminado de ejecutar se elimina la tabla que hay encima de la pila.

    La ejecución falla (devuelve FAILED) si la ejecución del nodo hijo falla, en caso contrario devuelve COMPLETED.
  - **PopContext** : Elimina la tabla de la cima de la pila del contexto, y devuelve COMPLETED si tiene éxito. La ejecución falla si la pila está vacía y en este caso se devuelve FAILED.

- **ExportSymbolNode** : Tiene 2 campos de tipo String que representan el nombre de 2 variables. Este nodo copia el valor de la variable 1 de la tabla de la cima de la pila a la tabla que hay justo debajo de la de la cima de la pila. Si el valor del campo 1 es la cadena vacía entonces se copia con el mismo nombre de variable, sino se copia con el nombre de variable que está almacenado como valor en el campo 2.

  Veamos un ejemplo en el que se quiere copiar la variable N3 en la variable N4 (figura 3.14) y cuando se quiere copiar con el mismo nombre (figura 3.15): La ejecución falla (devuelve FAILED) si la pila tiene 1 tabla o menos, en caso contrario devuelve COMPLETED.

  Este nodo es útil si antes de eliminar la tabla de la cima de la pila del contexto, queremos guardar algún dato almacenado en dicha tabla.

- **Nodos de estructuras de control** : Como su propio nombre indica, son los nodos encargados de implementar estructuras de control como

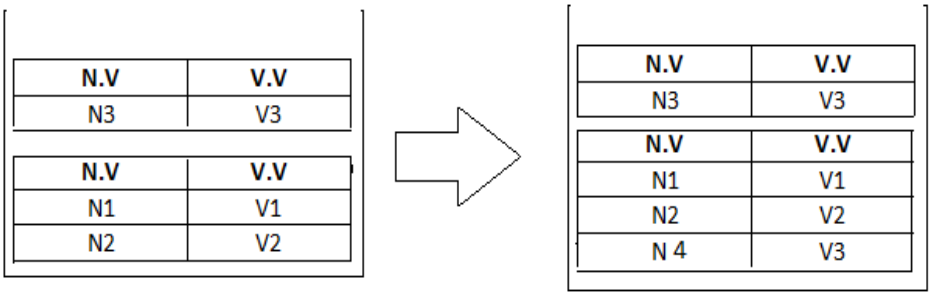


Figura 3.14: Exportar símbolo con un nombre distinto

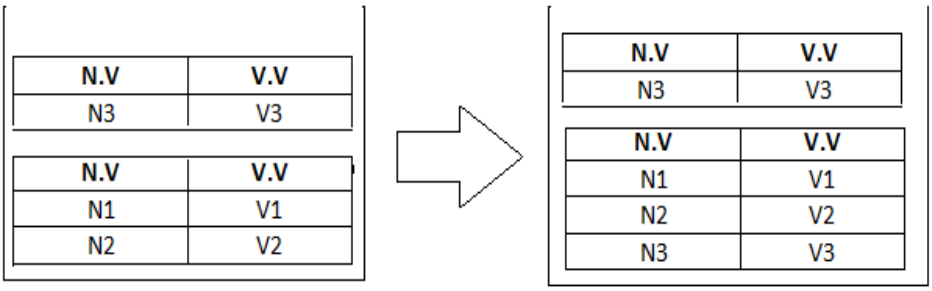


Figura 3.15: Exportar símbolo con el mismo nombre

por ejemplo el for, el while y el switch. Este tipo de nodos hereda de un nodo más general : `Composite`.

El nodo `Composite` se encarga de ejecutar una lista de nodos hijo hasta que uno de ellos falle. La ejecución tiene éxito (devuelve COMPLETED) si se consiguen ejecutar con éxito todos los nodos hijos. En caso contrario, en el momento en el que la ejecución de un nodo hijo falle, ya no se ejecutan más, informamos al BTExecutor del número de nodos que no hemos ejecutado y se devuelve FAILED.

Tiene un método para añadir un nodo a la lista de nodos hijo.

Por último tiene un método que calcula el número de nodos que no hemos llegado a ejecutar en el caso que la ejecución fallara antes de llegar al final de la lista. Dicha cantidad de nodos es la que se le pasa al BTExecutor cuando el método `execute()` falle como se mencionó anteriormente.

Los nodos que pertenecen a la categoría de estructuras de control son los siguientes:

- **Sequential** : Hereda del nodo Composite y se ejecuta de la misma manera que se explicó anteriormente cuando se habló de la

ejecución del nodo Composite es decir, ejecuta todos los nodos de la lista de nodos hijo. Si todos se ejecutan con éxito devuelve COMPLETED. Si alguno de los nodos falla su ejecución, se para la ejecución de los nodos de la lista y devuelve FAILED.

- **Or** : Nodo que hereda del Composite. Su finalidad es la de ir ejecutando los nodos de la lista de nodos hijo hasta encontrar uno que acabe con éxito, en ese momento finaliza la ejecución de los nodos de la lista y se devuelve COMPLETED. Si llega al último nodo y éste no se ejecuta con éxito se devuelve FAILED.
- **Random** : Ejecuta un nodo al azar de la lista de nodos hijo. Devuelve el resultado de la ejecución de dicho nodo.
- **If** : Implementa el comportamiento de la estructura If y contiene 3 nodos (el nodo condición, el nodo para la parte del Else y otro para la parte del Then). La funcionalidad de este nodo es la de ejecutar el nodo de la condición y dependiendo del resultado de la comparación se ejecuta alguno de los otros 2 nodos.
  Los pasos de la ejecución de este nodo son los siguientes:
  1. Se ejecuta el nodo de condición. Si la ejecución falla se devuelve FAILED (esto nunca va a pasar ya que un nodo condición siempre tiene éxito).
  2. Si la comparación se cumple (el valor de `condResult` en el contexto es `true`) vamos al paso 3. En otro caso, vamos al paso 4.
  3. Se ejecuta el nodo correspondiente a la parte del `Then`. Se devuelve el resultado de esa ejecución.
  4. Se ejecuta el nodo correspondiente a la parte del `Else`. Se devuelve el resultado de esa ejecución.
- **Switch** : Ejecuta el nodo de la lista de nodos hijo que se encuentra en la posición indicada por el valor de la variable `numInsSwitch` que se encuentra almacenada en el contexto. Devuelve el resultado de la ejecución de dicho nodo.
- **DoWhile** : Implementa el comportamiento de la estructura DoWhile. Hereda del Composite y también tiene un nodo condición. La funcionalidad de este nodo es la de ejecutar la lista de nodos hijos mientras que se cumpla una condición que se especifica mediante un nodo condición.

  Los pasos de ejecución de este nodo son los siguientes :
  - 1)Se ejecutan la lista de nodos hijo. Si la ejecución de alguno de los nodos hijo falla, se para la ejecución de los nodos de la lista y se devuelve FAILED.

- 2)Se ejecuta el nodo de la condición.
- 3)Si la condición se cumple, volvemos al paso 1.
- 4)Devuelve COMPLETED.

- **While** : Implementa el comportamiento de la estructura While. Hereda del nodo Composite. Se comporta igual que el DoWhile excepto que primero se comprueba la condición y luego se ejecuta la lista de nodos hijo.
- **For** : Implementa el comportamiento de la estructura For. Hereda del nodo Composite. La funcionalidad de este nodo es la de ejecutar la lista de nodos hijo un número de veces determinado.
  Los pasos de ejecución de este nodo son los siguientes:
  1. Se inicializa una variable auxiliar a 0 (aunque esto lo hace automáticamente el bucle for).
  2. Si el valor de dicha variable ha alcanzado el número de veces que se tiene que ejecutar la lista (valor que está almacenado en la variable `numIter` en el contexto) vamos al paso 5.
  3. Se ejecuta la lista de nodos hijo. Si alguna falla su ejecución, paramos la ejecución de la lista de nodos y devolvemos FAILED.
  4. Se aumenta en 1 el valor de la variable (aunque esto lo hace automáticamente el bucle for).
  5. Se devuelve COMPLETED.
- **Finally** : Implementa el comportamiento de la estructura Try-Finally. Tiene 2 nodos : uno para la parte del try y otro para la parte del finally. La funcionalidad de este nodo es la de ejecutar un nodo hijo y sin importar si la ejecución tiene éxito o no, ejecutamos otro nodo hijo.
  Los pasos de ejecución de este nodo son los siguientes:
  1. Se ejecuta el nodo Try.
  2. Se ejecuta el nodo Finally sin importar si la ejecución del nodo Try ha tenido éxito o no.
  3. Se devuelve el resultado de ejecutar el nodo Try.

- **Otros nodos** : En esta categoría entran otros nodos básicos que se han implementado y que realizan otras tareas básicas.

  Los nodos que pertenecen a esta categoría son los siguientes:

  - **Completed** : La ejecución de este nodo termina de forma satisfactoria automáticamente.

- **Failed** : La ejecución de este nodo termina de forma insatisfactoria automáticamente.
- **DecrementAction** : Le resta al valor de una variable almacenada en el contexto una cierta cantidad.
- **IncrementAction** : Le suma al valor de una variable almacenada en el contexto una cierta cantidad.
- **PrintAction** : Muestra un mensaje a través del Log (en la siguiente sección se hablará del Log).
- **PrintLastError** : Muestra a través del Log el último error ocurrido el cuál está almacenado en la variable `Last Error` en el contexto.
- **ReadAction** : Muestra a través del Log el nombre de una variable y su valor, estando ambos almacenados en el contexto.
- **ShowFinalValue** : Muestra a través del Log el resultado final de la ejecución de un árbol de comportamiento. Dicho resultado está almacenado en el contexto en la variable que se le pasa por parámetro a la constructora de este nodo.
- **WriteAction** : Almacena en el contexto una variable con un valor. Ambas cosas se le pasa por parámetro a la constructora de este nodo.

3.4. Ejemplos

En esta sección vamos a describir 2 ejemplos de árboles de comportamiento complejos creados a partir de los nodos básicos descritos anteriormente.

Para cada ejemplo, describiremos el contenido del contexto y el árbol de comportamiento. El resultado de la ejecución se mostrará mediante la interfaz gráfica creada.

3.4.1. Ejemplo 1

Explicación

En este primer ejemplo vamos a construir un árbol de comportamiento encargado de calcular el factorial (productorio) de un número y de mostrarlo a través de la interfaz gráfica.

Contenido del contexto

Para llevar a cabo el cálculo del factorial de un número necesitamos tener almacenadas en el contexto las siguientes variables:

- **stopKeyWhile** : Valor para el cuál termina el bucle. En este caso vale 0.
- **key3** : Valor del que se quiere calcular el factorial. Para este ejemplo, vale 5.
- **finalResultProd** : Variable donde se va almacenando el resultado de las multiplicaciones. Al final, contendrá el valor del factorial. Al principio vale 1.

Árbol de comportamiento

El árbol de comportamiento es de la Figura 3.16.

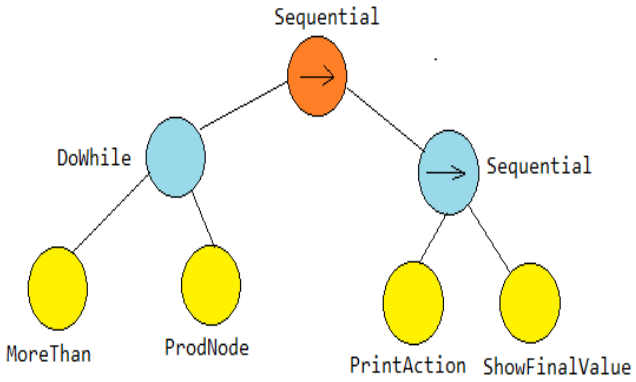


Figura 3.16: Ejemplo 1

Explicación del árbol de comportamiento

El proceso de ejecución del árbol de comportamiento es el siguiente:

- El nodo `DoWhile` tiene como nodo condición un nodo `MoreThan` que mira si el valor de la variable `key3` es mayor que el de la variable `stopKeyWhile1` y como lista de nodos hijo una lista con un único nodo, un `ProdNode`, el cuál primero multiplica el valor de la variable `key3` al valor de la variable `finalResultProd` y luego resta 1 al valor de `key3`.
- El nodo `PrintAction` muestra el mensaje `RESULTADO DEL PRODUCTORIO`.

- El nodo `ShowFinalValue` muestra el valor final de la variable `finalResultProd`.

Resultado de la ejecución del árbol de comportamiento

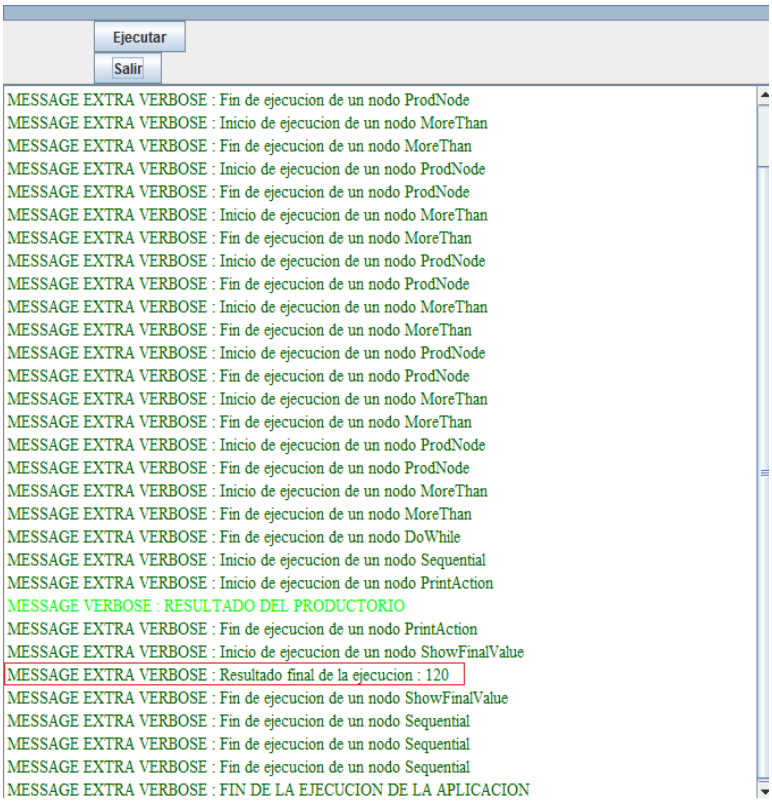


Figura 3.17: Ejecución del ejemplo 1

El resultado de la ejecución del árbol de comportamiento es el de la Figura 3.17.

En la figura no se muestra toda la ejecución ya que en la parte donde se van mostrando los mensajes es demasiado pequeña como para que quepan todos y por eso aparece una barra de desplazamiento vertical.

Como podemos observar , efectivamente, se muestra 120 (factorial de 5) como el valor final de la variable.

3.4.2. Ejemplo 2

Explicación

En este primer ejemplo vamos a construir un árbol de comportamiento

encargado de calcular el sumatorio desde 1 hasta cierto número y de mostrarlo a través de la interfaz gráfica.

Contenido del contexto

Para llevar a cabo el cálculo del sumatorio necesitamos tener almacenadas en el contexto las siguientes variables:

- **stopKeyWhile** : Valor para el cuál termina el bucle. En este caso vale 0.
- **key3** : Valor del que se quiere calcular el sumatorio. Para este ejemplo, vale 5.
- **finalResultSum** : Variable donde se va almacenando el resultado de las sumas. Al final, contendrá el valor del sumatorio. Al principio vale 0.

Árbol de comportamiento

El árbol de comportamiento es de la Figura 3.18.

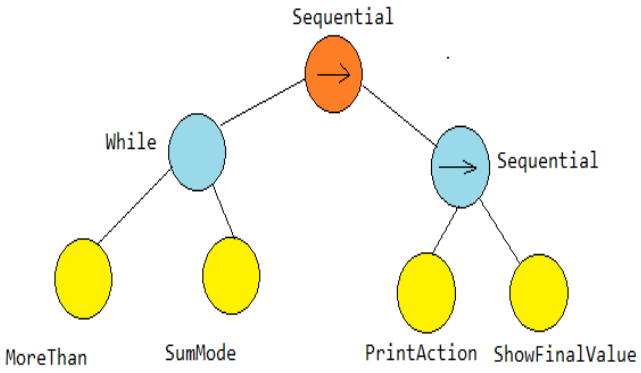


Figura 3.18: Ejemplo 2

Explicación del árbol de comportamiento

El proceso de ejecución del árbol de comportamiento es el siguiente:

- El nodo `While` tiene como nodo condición un nodo `MoreThan` que mira si el valor de la variable `key3` es mayor que el de la variable `stopKeyWhile1` y como lista de nodos hijo una lista con un único nodo, un `SumNode`, el cuál primero suma el valor de la variable `key3` al valor de la variable `finalResultSum` y luego resta 1 al valor de `key3`.
- El nodo `PrintAction` muestra el mensaje `RESULTADO DEL SUMATORIO`.

- El nodo `ShowFinalValue` muestra el valor final de la variable `finalResultSum`.

Resultado de la ejecución del árbol de comportamiento

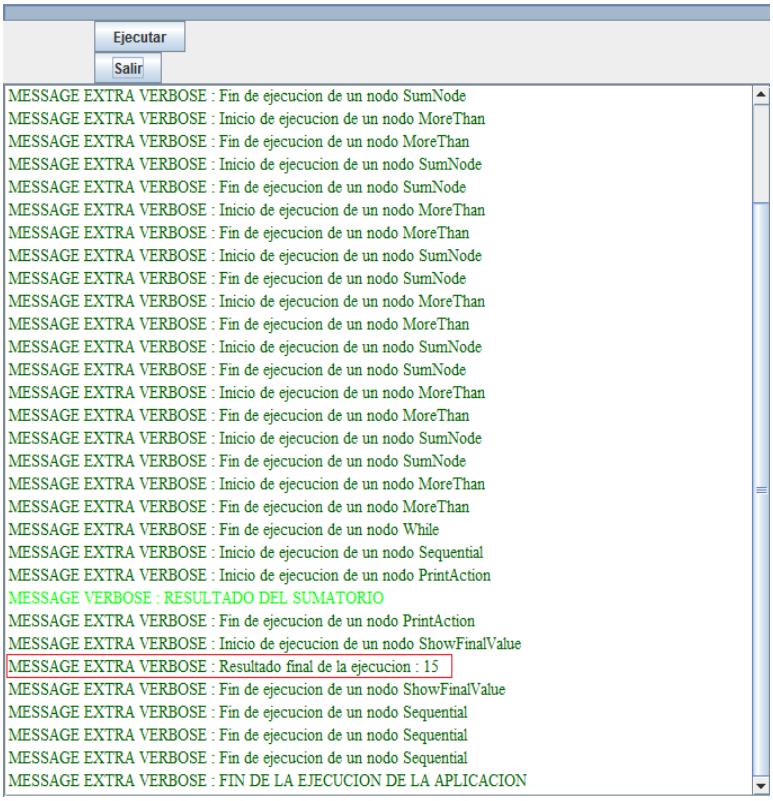


Figura 3.19: Ejecución del ejemplo 2

El resultado de la ejecución del árbol de comportamiento es el de la Figura 3.19.
En la figura no se muestra toda la ejecución ya que en la parte donde se van mostrando los mensajes es demasiado pequeña como para que quepan todos y por eso aparece una barra de desplazamiento vertical.
Como podemos observar , efectivamente, se muestra 15 (sumatorio desde 1 hasta 5) como el valor final de la variable.

Capítulo 4

Framework de Validación

Resumen: En este capítulo hablaremos de la segunda parte del proyecto la cuál es la encargada del `Framework de Validación`. En la sección 4.1 vamos a ver un ejemplo de aplicación de los conceptos que se van a tratar en este capítulo. En el apartado 4.2 hablaremos tanto de la ampliación de la lista de nodos implementados en la primera parte del proyecto (los nodos de los que se habló en el anterior capítulo) como de la ampliación de los aspectos relacionados con el Log. También hablaremos de una nueva versión de la interfaz gráfica creada para ver el progreso del proceso de validación de una práctica. Para acabar, en esa misma sección trataremos todo lo relacionado con la compilación y ejecución de código fuente, y de todo lo relacionado con los ficheros y directorios. En la sección 4.3 hablaremos de los aspectos más importantes de la implementación de todos esos elementos. Por último, en la sección 4.4, construiremos varios nodos avanzados a partir de los anteriores, los cuáles nos serán útiles para la parte de la aplicación encargada de la validación de prácticas que se explicará en el próximo capítulo.

4.1. Introducción

Antes de pasar a explicar los elementos de los que consta el `Framework de Validación` vamos a ver un ejemplo sencillo de práctica y que es lo que debe hacer el validador.

4.1.1. Código de la práctica

La práctica es la siguiente:

```
#include <iostream>

int main() {
    std::cout << \"Hola mundo\" << std::endl;
}
```

Como podemos ver, es un ejemplo muy sencillo pero nos ayudará a comprender lo que debe hacer el validador. Lo único que hace es mostrar por pantalla "Hola mundo". Este es el típico programa que se usa como primer ejemplo cuando se estudia por primera vez un lenguaje de programación.

4.1.2. Etapas del proceso de validación

Los pasos que tiene que hacer el validador para comprobar si la práctica está bien implementada pueden ser los siguientes:

1. Compilar la práctica.

2. Ejecutar la práctica.

3. Comprobar que la práctica hace lo que tiene que hacer.

4.1.3. Explicación del proceso de validación

A grandes rasgos y sin meternos mucho en como se implementaría (esto se explica más adelante en este mismo capítulo), vamos a explicar lo que pasa en cada etapa del proceso de validación. Para ello, supongamos que el código anterior se encuentra en un fichero llamado "prueba.cpp", que el fichero ejecutable queremos que se llame "a.out", que la salida del programa se redirige al fichero "salida.txt y que la salida que debe dar se encuentra en el fichero "salidaEsperada.txt".

1. **Compilación** : El validador debe ejecutar la orden `gcc <ruta del ‘‘prueba.cpp’’><ruta del ‘‘a.out’’>` la cuál compila el fichero "prueba.cpp" y crea como fichero ejecutable "a.out".

2. **Ejecución** : El validador debe ejecutar la orden `<ruta del ‘‘a.out’’ (sin la extensión)>` y redireccionar la salida a <ruta del "salida.txt">la cuál ejecuta el fichero "a.out" y escribe `Hola mundo` en el fichero "salida.txt".

3. **Comprobación** : El validador compara el contenido de los ficheros "salida.txt" y "salidaEsperada.txt" mediante un programa de comparador de ficheros y muestra cuánto se parecen (en %). En este caso, como la práctica es simple, los ficheros van a tener el mismo contenido.

En este caso, el fichero "salidaEsperada.txt'debe contener `Hola mundo`.

El árbol de comportamiento es algo parecido al que se muestra en la figura 4.1 (a los nodos se les llamará de otra manera):

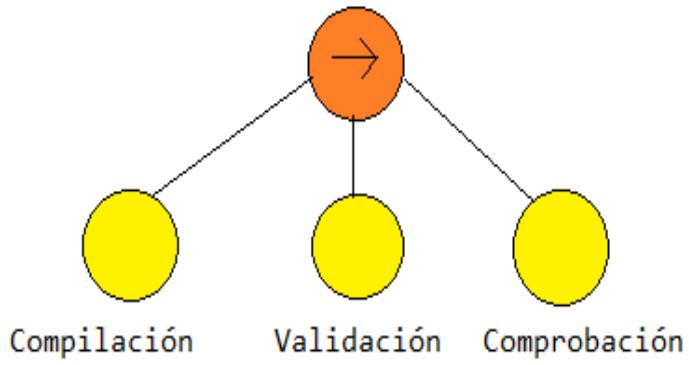


Figura 4.1: Árbol de comportamiento de la práctica

4.2. Explicación de los elementos del framework

4.2.1. Log

En esta segunda parte del proyecto se ha ampliado las operaciones que se pueden realizar con el Log.
Dichas operaciones ahora son las siguientes:

- Mostrar mensajes de información.
- Mostrar mensajes súper detallados.
- Mostrar mensajes detallados.
- Mostrar resultados.
- Mostrar avisos.
- Mostrar errores.
- Mostrar errores fatales.

- Mostrar errores críticos.
- Mostrar mensaje con el nombre de una etapa de ejecución.

Además, también se han creado nuevas salidas (`Output`) por donde se mostrarán dichos mensajes:

- **XMLOutput** : Se usa la salida estándar (`System.out`) y la salida de error (`System.err`) para mostrar los mensajes con formato XML. También permite cambiar la salida por donde se muestran los mensajes.
- **ConsoleOutput** : Igual que la anterior pero sin formato XML. Es decir, se muestran los mensajes tal y como llegan.
- **DobleOutput** : Redirige los mensajes a 2 salidas (`Output`) diferentes.
- **TextAreaOutput** : Se usa un `JEditorPane` de la librería de Java `swing` para mostrar los mensajes.
- **ProxyOutputValidador** : Salida que redirige los mensajes generados por los nodos de la parte del framework del `EagerBT` al Log creado en este framework. Cuando el nodo `PrintAction` mostraba un mensaje por el Log, lo mostraba a través del Log creado en el framework del `EagerBT`, y cuando un nodo `CriticalError`, por ejemplo, manda un mensaje al Log lo manda al Log de esta parte: y como esos 2 Log son diferentes nunca se van a mostrar ambos mensajes por la misma salida. Es por esto último por lo que se ha tenido que crear este tipo de salida para redirigir los mensajes de todos los nodos hacia la misma salida.

4.2.2. Interfaz gráfica : Versión 2

Esta segunda versión de la interfaz (y versión definitiva) es la que se ha implementado para ser usada posteriormente en la aplicación encargada de la validación de prácticas.

La nueva interfaz gráfica tiene el aspecto que se muestra en la figura 4.2.

Está compuesta por los siguientes elementos :

- Un campo de texto donde el estudiante especifica la ruta del zip donde se encuentran los ficheros fuente de la práctica implementada.
- Un botón para mostrar un cuadro de diálogo donde el usuario puede elegir la ruta del zip.

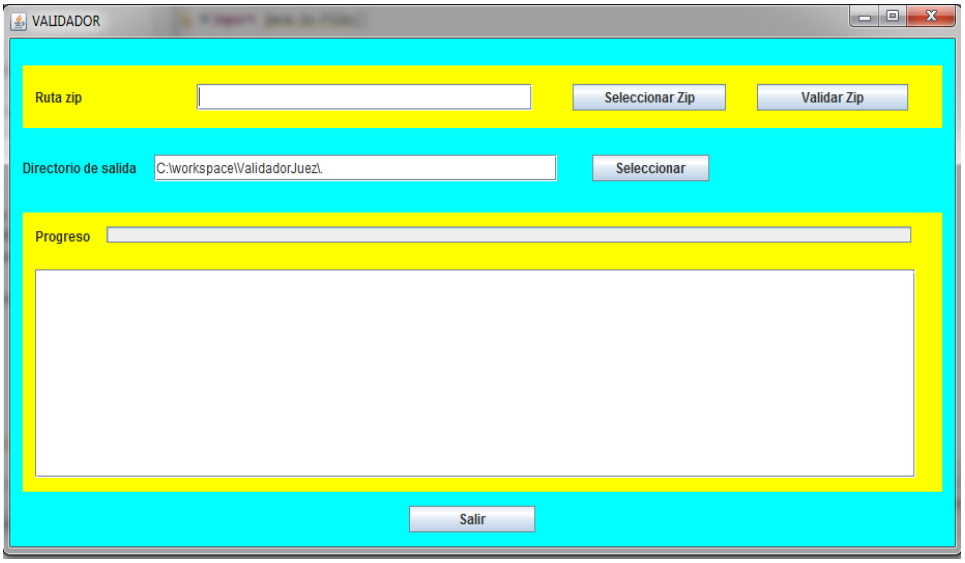


Figura 4.2: Interfaz gráfica para la validación de la práctica

- Un campo de texto donde el estudiante especifica el directorio de salida. Por defecto, contiene el directorio actual. item Un botón para mostrar un cuadro de diálogo donde el usuario puede elegir el directorio de salida.
- Un botón para salir de la aplicación.
- Un botón para comenzar la validación de la práctica.
- Un panel de información (del tipo `TextAreaOutput` para que sea un Log) donde van apareciendo los mensajes que van mandando los nodos.
- Una barra de progreso (valor mínimo : 0 y valor máximo : número total de nodos del árbol de comportamiento) para ir viendo el tanto por ciento de los nodos ejecutados.

4.2.3. Nodos

Para este nuevo framework se han creado los siguientes nuevos tipos de nodos:

- **Nodos de entrada/salida** : Nodos encargados de ejecutar operaciones de entrada y salida. En particular, ejecutan operaciones sobre ficheros. Dentro de este tipo se han creado los siguientes nodos:

  - **CreateDirNode** : Crea un directorio dado por una ruta.
  - **FileSize** : Calcula el tamaño de un archivo en bytes.
  - **RemoveDirNode** : Elimina un directorio dado por una ruta.
  - **UnzipFileNode** : Descomprime un zip en un directorio dado.
  - **FindFiles** : Recorre un directorio y añade al contexto la lista de ficheros de ese directorio.

- **Nodos de mensajes** : Nodos encargados de mostrar mensajes a través del Log. Dentro de este tipo se han creado los siguientes nodos:

  - **CriticalError** : Muestra a través del Log un error crítico.
  - **Error** : Muestra a través del Log un error normal.
  - **ExtraVerbose** : Muestra a través del Log un mensaje súper detallado.
  - **FatalError** : Muestra a través del Log un error fatal.
  - **Verbose** : Muestra a través del Log un mensaje detallado.
  - **Warning** : Muestra a través del Log un aviso.
  - **MatchedFiles** : Muestra a través del Log un mensaje que indica lo que se parecen 2 ficheros.
  - **StartStage** : Muestra a través del Log la etapa en la que nos encontramos en el proceso de ejecución del árbol. En la aplicación encargada de la validación se usa para mostrar la etapa en la que nos encontramos en el proceso de validación de una práctica.

4.2.4. Manejo de ficheros

Como la aplicación desarrollada es un validador de prácticas el cuál trabaja con ficheros, necesitamos implementar una clase que contenga métodos relacionados con el manejo de ficheros.

Nuestro framework permite las siguientes operaciones sobre ficheros:

- Dada la ruta de un fichero, obtener el nombre de dicho fichero.
- Dado el nombre de un fichero, devolver la extensión de dicho fichero.
- Dada la ruta o el nombre de un fichero, quitar la extensión de dicho fichero.
- Leer el contenido de un fichero y almacenarlo en una cadena de texto.
- Escribir una cadena de texto en un fichero.

- Borrar un fichero.
- Comprobar si 2 ficheros tienen el mismo contenido.
- Añadir el contenido de un fichero al final del contenido de otro fichero.

4.2.5. Compilación de código fuente

Al ser la aplicación desarrollada un validador de práctica es fundamental la parte de la compilar los ficheros fuente (clases) de una práctica. Distinguimos varios tipos de compilación dependiendo del lenguaje de programación sobre el que está escrito los ficheros de la práctica:

- **Compilador C** : Para compilar una aplicación en C se necesitan los siguientes datos:
  - Compilador de C : `gcc`.
  - Una cadena formada por la concatenación de la definición de los símbolos, la parte de la definición de las opciones, la ruta del fichero de salida y la ruta de los ficheros fuente.
  - Directorio de trabajo.

  La compilación se realiza creando un objeto de la clase `Ejecuta` (se explicará más adelante) pasándole como parámetros los datos anteriores.
- **Compilador C++** : Para compilar una aplicación en C se necesitan los siguientes datos:
  - Compilador de C++ : `g++`.
  - Una cadena formada por la concatenación de la definición de los símbolos, la parte de la definición de las opciones, la ruta del fichero de salida y la ruta de los ficheros fuente.
  - Directorio de trabajo.

  La compilación se realiza creando un objeto de la clase `Ejecuta` (se explicará más adelante) pasándole como parámetros los datos anteriores.
- `Compilador Java` : Para compilar una aplicación Java se necesitan los siguientes datos:
  - Compilador Java.
  - Ficheros fuente.
  - Directorio de los ficheros fuente.
  - Directorio donde se encuentran los .jar que se necesiten.
  - Directorio de salida.

4.2.6. Ejecución de código fuente

Al ser la aplicación desarrollada un validador de práctica es fundamental la parte de la ejecución de las clases que forman una práctica. Distinguimos varios tipos de ejecución dependiendo del lenguaje de programación sobre el que está escrito los ficheros de la práctica:

- Ejecución de código Python.
- Ejecución de código Java.
- Ejecución de código nativo.

A continuación vamos a hacer un repaso general de cada uno de esos tipos de ejecución :

- **Ejecución Java** : Para ejecutar código Java desde la ventana de comandos se necesitan los siguientes datos:
  - Ejecutador de Java (`java.exe` si estamos en Windows y si no, `java`).
  - Si existen librerías auxiliares o jar, la ruta de estos.
  - La ruta de la carpeta de los .class.
  - El nombre del .class de la clase que contiene el `main`.
  - Los argumentos de entrada de esa clase.
  - El directorio de trabajo.
- **Ejecución Python** : Para ejecutar código Python desde la ventana de comandos se necesitan los siguientes datos:
  - Ejecutador de Python.
  - El archivo .py a ejecutar.
  - Los argumentos de entrada de ese archivo.
  - El directorio de trabajo.
- **Ejecución código nativo** : Para ejecutar código nativo desde la ventana de comandos se necesitan los siguientes datos:
  - Aplicación a ejecutar (por ejemplo un .exe).
  - Los parámetros de entrada de la aplicación.
  - El directorio de trabajo.

En la siguiente sección veremos como se traduce la manera de ejecutar código desde la ventana de comandos a la manera de ejecutarlo desde nuestro framework.

4.2.7. Manejo de directorios

Como la aplicación también trabaja con directorios, necesitamos implementar una clase que contenga métodos relacionados con el manejo de directorios.

Nuestro framework permite las siguientes operaciones sobre directorios:

- Devolver una lista con todos los ficheros de un directorio y de todos los subdirectorios de ese directorio.
- Devolver una lista con todos los ficheros de un directorio (sin incluir los de sus subdirectorios)
- Devolver los subdirectorios de un directorio.
- Devolver el número de ficheros que contiene un directorio.
- Eliminar un directorio y todo su contenido.
- Crear un directorio temporal y devolver su ruta.
- Crear un directorio normal (no temporal).
- Comprobar si existe un directorio.
- Devolver una cadena con la ruta a partir de los nombres de las rutas parciales.

4.2.8. Manejo de archivos zip

Otra de las acciones que debe realizar nuestra aplicación es la de poder descomprimir un .zip . Por lo tanto se ha implementado una clase que dada la ruta de un archivo zip, lo descomprime en otra ruta.

4.3. Implementación

En esta sección vamos a hablar de los aspectos más importantes de la implementación de cada uno de los aspectos de los que se habló anteriormente.

4.3.1. Log

De la parte del Log, la parte más importante de la implementación es la que se refiere a los diferentes tipos de salida (`Output`):

- **ConsoleOutput** : Los mensajes de error se muestran por `System.err` y el resto de mensajes por `System.out`.

- **DobleOutput** : Esta salida está formada por 2 salidas a la vez. Cada vez que se manda un mensaje, se le manda a las 2 salidas a la vez.
- **ProxyOutputValidador** : Los mensajes se muestran a través del Log.
- **XMLOutput** : Tiene un campo que indica la salida por la que se van a mostrar los mensajes en formato XML. Para pasar un mensaje a formato XML dicho mensaje se transforma en `<tipo-mensaje>mensaje </tipo-mensaje>`.
- **TextAreaOutput** : Tiene un campo del tipo `JEditorPane` al que se configura para tener mensajes en formato HTML para que cada tipo de mensaje se muestre de un color distinto y poderlos distinguir más fácilmente.

4.3.2. Interfaz gráfica : versión 2

Los aspectos más importantes relacionados con la creación y funcionamiento de la interfaz gráfica son los siguientes:

- Cuando se ejecuta la aplicación del validador se crea el árbol de comportamiento para la validación de la práctica y un objeto de la clase `ValidatorWindow` el cuál representa la interfaz gráfica y al que se pasa como parámetro el árbol.
- En la constructora de la interfaz gráfica además de crear la parte visual, se asigna como salida del Log el panel de información de la interfaz gráfica donde se van mostrando los mensajes.
- Si el usuario pulsa el botón `Validar Zip` y no ha especificado ni la ruta del zip que contiene los ficheros de la práctica, ni ha especificado el directorio de salida aparece un mensaje de error por el panel de información. En caso contrario, se crea un nuevo thread en el que se ejecutan las siguientes acciones:
  - Se desactiva el botón `Validar zip`.
  - Se limpia el panel de información.
  - Se crea un contexto en el que se añaden las siguientes variables:
    - **Zip Ejercicio** : Tiene como valor la ruta del fichero zip especificado por el usuario.
    - **debugMode** : Indica si se quiere ejecutar el árbol en modo DEBUG.
    - **Output Dir** : Tiene como valor el directorio de salida especificado por el usuario.
    - **jargs.jar** : Tiene como valor la ruta donde se encuentra `jargs.jar`. En este caso, la ruta es `.//jargs.jar`.

- **Max Size** : Tiene como valor el tamaño máximo permitido para el fichero zip.

- Se crea un nuevo árbol de comportamiento llamado `validador` a partir del árbol de validación de la práctica.

  Este nuevo árbol es el de la figura 4.3. El último nodo depende del

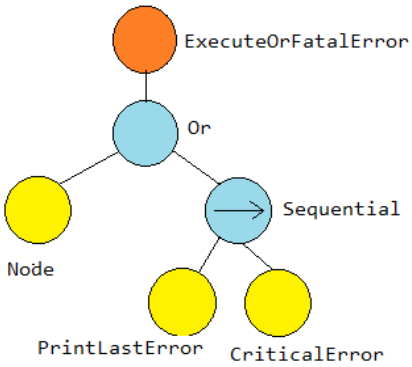


Figura 4.3: Nodo Árbol `validador`

  valor de la variable `debugMode` almacenada en el contexto. Si vale `true` es un nodo `True` y si vale `false` es un nodo `RemoveDirNode` que elimina el directorio de trabajo. El comportamiento de este árbol es el siguiente : se ejecuta el árbol de validación y después se ejecuta el otro nodo hijo sin importar si el otro ha tenido éxito o no.

- Se crea un nuevo objeto de la clase BTExecutor y se le añade a la lista observadores esta clase (ya que esta implementa la clase Observer).
- Se asigna al atributo `numNodes` de la clase el número total de nodos del árbol `validador`.
- Se ejecuta el árbol `validador`. Si la ejecución tiene éxito, se muestra en el panel de información un mensaje súper importante indicando que la práctica esta bien implementado. Si la ejecución ha fallado, se muestra un error fatal indicando que la práctica está mal implementada.
- Se vuelve a activar el botón `Validar zip`.

- En total se ejecutan 2 threads En uno se ejecuta el árbol `validador`. El otro se usar para cambiar la barra de progreso y para mostrar los mensajes en el panel de información. Si solo hubiera 1 thread no se podría ejecutar el árbol a la vez que se cambia la barra de progreso y se muestran los mensajes.

- Como la clase `ValidatorWindow` implementa la interfaz Observer, contiene los siguientes métodos:

  - **advanceNode()** : No realiza ninguna acción.
  - **onStartNodeExecution()** : No realiza ninguna acción.
  - **skipNodes(numNodes)** : Avanza la barra de progreso `numNodes` unidades.
  - **repeatNodes(numNodes)** : Retrocede la barra de progreso `numNodes` unidades.
  - **onStartTreeExecution()** : Limpia el panel de información, reinicia la barra de progreso y especifica el valor máximo de la barra de progreso el cuál se pone al número de nodos del árbol `validador`.
  - **onEndTreeExecution()** : Activa el botón `Validar zip`.
  - **onEndNodeExecution()** : Avanza la barra de progreso una unidad.

4.3.3. Nodos

Los aspectos más importantes de la implementación de los nodos son los siguientes :

- **Nodos de entrada y salida :**

  - **CreateDirNode** : Tiene un campo para el nombre del directorio que se quiere crear. Cuando se ejecuta, llama al método `create(<nombre-dir>)` de la clase `Folder` creada. Devuelve COMPLETED si se ha podido crear el directorio y FAILED si no se ha podido.
  - **FileSize** : Tiene un campo para el nombre del fichero cuyo tamaño se quiere calcular. Cuando se ejecuta, primero se crea un objeto de la clase `File` (librería `java.io` de Java) y se llama al método `length()` de dicha clase.
  - **FindFiles** : Tiene un campo para el nombre del directorio del que se quiere obtener la lista de nombre de todos los ficheros que contiene. Cuando se ejecuta, llama al método de la clase `Folder` (creada en este framework) que realiza una búsqueda recursiva de los ficheros de un directorio. Cuando ha obtenido una lista con los nombres de todos los ficheros la añade al contexto como valor de la variable `Files`. La ejecución devuelve siempre COMPLETED.

- **RemoveDirNode** : Tiene un campo para el nombre del directorio que se quiere eliminar. Cuando se ejecuta, primero manda un mensaje al Log avisando que se va a eliminar un directorio y después llama al método de la clase `Folder` encargado de eliminar un directorio. La ejecución devuelve COMPLETED si se ha conseguido eliminar el directorio con éxito.
- **UnzipFileNode** : Tiene un campo para el nombre del fichero zip que se quiere descomprimir y otro para indicar el directorio de destino. Cuando se ejecuta, se llama al método de la clase `Unzip` encargado de descomprimir un fichero zip. Si se ha podido descomprimir con éxito se devuelve COMPLETED y si no, se manda al Log un mensaje de error fatal indicando que ha habido un fallo al descomprimir y devuelve FAILED.

- **Nodos de mensajes :**
  - **CriticalError** : Manda al Log un mensaje de error crítico y falla (devuelve FAILED). Además escribe en el contexto una variable `CriticalError` con valor `True`.
  - **Error** : Manda al Log un mensaje de error y tiene éxito (devuelve COMPLETED).
  - **ExtraVerbose** : Manda al Log un mensaje súper detallado y tiene éxito (devuelve COMPLETED).
  - **FatalError** : Manda al Log un mensaje de error fatal y falla (devuelve FAILED).
  - **MatchedFiles** : Manda al Log un mensaje indicando cuanto se parecen 2 ficheros de texto y tiene éxito (devuelve COMPLETED).
  - **StartStage** : Manda al Log un mensaje indicando el inicio de una nueva etapa del proceso de ejecución y tiene éxito (devuelve COMPLETED).
  - **Verbose** : Manda al Log un mensaje detallado y tiene éxito (devuelve COMPLETED).
  - **Warning** : Manda al Log un mensaje de aviso y tiene éxito (devuelve COMPLETED).

4.3.4. Compilación de código fuente

Para la compilación de los distintos ficheros de código fuente de la práctica que se quiere validar, se han implementado las siguientes clases para cada uno de los tipos de compilación:

- **CCompilation** : Interfaz que representa una compilación de C con el compilador de C.

- **CCompiler** : Clase encargada de buscar en el sistema el compilador de C. Esta clase contiene lo siguiente:
  - Un atributo de tipo `CCompilation` para almacenar el compilador de C del sistema.
  - Un método para obtener el compilador de C. Realiza los siguientes pasos:
    - Si el atributo que almacena el compilador es distinto de NULL, devuelve una copia del atributo.
    - Crea una carpeta temporal y crea en dicha carpeta un fichero fuente lo suficientemente sencillo como para comprobar la existencia de un compilador de C.
    - Crea una variable auxiliar de tipo `GNUCCompiler` (se explicará más adelante) y se le añade como fichero fuente el fichero que hemos creado y como fichero de salida uno cualquiera.
    - Llama al método compilar de esa variable. Si la compilación no tiene éxito se pone esa variable a NULL.
    - Elimina el directorio temporal.
    - Asigna el valor de esa variable al atributo que almacena el compilador.
    - Devuelve la variable.
- **GNUCCompiler** : Clase que implementa la interfaz `CCompilation` y representa el compilador g++. Esta clase contiene lo siguiente:
  - Atributos que representan : una lista de ficheros fuente, nombre de fichero de salida, lista de definición de símbolos, lista de opciones, directorio de trabajo y salida del compilador (en formato cadena).
  - Métodos para especificar el valor de cada uno de los atributos.
  - Método que compila una aplicación usando el compilador gcc. Consiste en crear un objeto de la clase `Ejecuta` pasándole como parámetros los siguientes datos:
    - Nombre del compilador de C : "gcc".
    - Un vector de String con los siguientes elementos : las definiciones de símbolos, las opciones, el fichero de salida y los ficheros fuente.
    - Directorio de trabajo.
  - Método que devuelve la versión del compilador gcc. Consiste en crear un objeto de la clase `Ejecuta` pasándole como parámetros los siguientes datos:
    - Nombre del compilador de C : "gcc".
    - Un vector de String con el elemento "–version".

    - Directorio de trabajo : ".".

- **CPPCompilation** : Interfaz que representa una compilación de C++ con el compilador de C++.
- **CPPCompiler** : Clase encargada de buscar en el sistema el compilador de C++. Esta clase contiene lo siguiente:
    - Un atributo de tipo `CPPCompilation` para almacenar el compilador de C++ del sistema.
    - Un método para obtener el compilador de C++. Realiza los siguientes pasos:
        - Si el atributo que almacena el compilador es distinto de NULL, devuelve una copia del atributo.
        - Crea una carpeta temporal y crea en dicha carpeta un fichero fuente lo suficientemente sencillo como para comprobar la existencia de un compilador de C++.
        - Crea una variable auxiliar de tipo `GNUCPPCompiler` (se explicará más adelante) y se le añade como fichero fuente el fichero que hemos creado y como fichero de salida uno cualquiera.
        - Llama al método compilar de esa variable. Si la compilación no tiene éxito se pone esa variable a NULL.
        - Elimina el directorio temporal.
        - Asigna el valor de esa variable al atributo que almacena el compilador.
        - Devuelve la variable.
- **GNUCCompiler** : Clase que implementa la interfaz `CCompilation` y representa el compilador g++. Esta clase contiene lo siguiente:
    - Atributos que representan : una lista de ficheros fuente, nombre de fichero de salida, lista de definición de símbolos, lista de opciones, directorio de trabajo y salida del compilador (en formato cadena).
    - Métodos para especificar el valor de cada uno de los atributos.
    - Método que compila una aplicación usando el compilador g++. Consiste en crear un objeto de la clase `Ejecuta` pasándole como parámetros los siguientes datos:
        - Nombre del compilador de C++ : "g++".
        - Un vector de String con los siguientes elementos : las definiciones de símbolos, las opciones y el fichero de salida.
        - Directorio de trabajo.
    - Método que devuelve la versión del compilador g++. Consiste en crear un objeto de la clase `Ejecuta` pasándole como parámetros los siguientes datos:

    - Nombre del compilador de C++ : "g++".
    - Un vector de String con el elemento "–version".
    - Directorio de trabajo : ".".

- **JavaCompiler** : Clase encargada de buscar en el sistema el compilador de Java. Esta clase contiene lo siguiente:

  - Un atributo de tipo `JavaCompilation` para almacenar el compilador de Java del sistema.
  - Un método para obtener el compilador de Java. Realiza los siguientes pasos:
    - Si el atributo que almacena el compilador es distinto de NULL, devuelve una copia del atributo.
    - Crea una carpeta temporal y crea en dicha carpeta un fichero fuente lo suficientemente sencillo como para comprobar la existencia de un compilador de Java.
    - Crea una variable auxiliar de tipo `JavaCompilation` (se explicará más adelante) y se le añade como fichero fuente el fichero que hemos creado y como fichero de salida uno cualquiera.
    - Llama al método compilar de esa variable. Si la compilación no tiene éxito se pone esa variable a NULL.
    - Elimina el directorio temporal.
    - Asigna el valor de esa variable al atributo que almacena el compilador.
    - Devuelve la variable.

- **JavaCompilation** : Clase que representa una compilación en Java con el JDK encontrado en el sistema. Esta clase contiene lo siguiente:

  - Atributos que representan : una lista con los ficheros fuente, una lista con los directorios donde se encuentran dichos ficheros fuente, el directorio de salida y una lista con los directorios donde se encuentran los .jar que se necesitan.
  - Métodos para especificar los valores de cada uno de los atributos.
  - Método para obtener la versión del compilador de Java del sistema. Realiza los siguientes pasos :
    - Si no se encuentra el compilador de Java, devuelve un mensaje de error.
    - Crea un objeto de la clase `Ejecuta` al que se le pasan los siguientes datos:
      - Compilador de Java : "javac".
      - Un array de String con el elemento "-version".

        - ◇ Directorio de trabajo : ".".
      - Llama al método `ejecuta()` de dicho objeto para obtener la versión del compilador de Java.
  - Método para compilar una aplicación usando el compilador de Java. Realiza los siguientes pasos:
      - Crea un objeto `diagnostics` de tipo `DiagnosticCollector <JavaFileObject>` (`DiagnosticCollector` proporciona una manera fácil de coleccionar diagnósticos en una lista y `JavaFileObject` es una abstracción de ficheros para herramientas operando en ficheros fuentes y clases en Java).
      - Crea un objeto `fileManager` de la clase `StandardJavaFileManager` (gestor de ficheros Java) que se crea llamando al método `JavaCompiler.compiler.getStandardFileManager` (siendo el `JavaCompiler` que proporciona la librería Java `javax.tools`) al que se le pasa como parámetro la variable `diagnostics`.
      - Se le indica al `fileManager` el directorio de salida, los directorios con los ficheros fuente y los directorios con los .jar.
      - Consigue los ficheros de código llamando al método `getJavaFileObjectsFromStrings` del `fileManager` pasándole como parámetro los ficheros fuente.
      - Monta la tarea de validación del tipo `CompilationTask` de la librería `javax.tools.JavaCompiler` que se crea llamando al método `JavaCompiler.compiler.getTask` (el JavaCompiler de la librería de Java mencionada anteriormente) pasando como parámetros las variables `fileManager` y `diagnostics` y los ficheros de código.
      - Ejecuta la tarea.
      - Devuelve `true` si se ejecutado con éxito la tarea y `false` en caso contrario.

4.3.5. Ejecución de código fuente

Para la ejecución de los distintos ficheros de código fuente de la práctica que se quiere validar, se han implementado los siguientes métodos para cada uno de los tipos de ejecución :

- **Ejecución en Java** Para la ejecución de una práctica (aplicación) en Java se ha creado la clase `EjecutaJava` la cuál contiene lo siguiente :
  - Atributos para representar : el .class que contiene el método `main()`, los parámetros de entrada, el directorio de trabajo, los argumentos que se le pasan a la aplicación, la entrada a la aplicación (en forma de cadena o como fichero de entrada) y el fichero de salida.

- Métodos para especificar cada uno de los atributos.
- Método para ejecutar la aplicación. Se realizan los siguientes pasos:
  - Se busca la ruta del ejecutador de Java (`java.exe` en Windows o `java` en otro caso).
  - Se crea una cadena concatenando la ruta de los .jar que se necesiten, la ruta de la carpeta donde se encuentran los .class de la aplicación, el .class de la clase que contiene el método `main()` y los argumentos de entrada a ese `main()`.
  - Se crea un objeto de la clase Ejecuta a la que se le pasan la ruta del Ejecutador de Java, la cadena creada anteriormente y el directorio de trabajo.
  - Se especifica la entrada de la aplicación (tanto en formato cadena como el fichero de entrada), el fichero de salida, el fichero para los errores en los métodos correspondientes del objeto creado de la clase Ejecuta.
  - Se ejecuta dicho objeto pasándole como parámetro el tiempo limite para ejecutar la aplicación.

- **Ejecución en Python** Para la ejecución de una práctica (aplicación) en Python se ha creado la clase `EjecutaPython` la cuál contiene lo siguiente :
  - Atributos para representar : el fichero .py y la lista de parámetros de la aplicación.
  - Métodos para especificar cada uno de los atributos.
  - Método para ejecutar la aplicación. Se realizan los siguientes pasos:
    - Se crea un objeto de la clase Ejecuta a la que se le pasan como Ejecutador de Python "python", el fichero .py y el directorio de trabajo (siempre va a valer ".").
    - Se ejecuta dicho objeto pasándole como parámetro el tiempo limite para ejecutar la aplicación.

- **Ejecución** Para la ejecución de una práctica (aplicación) sin importar el lenguaje, se ha creado la clase `Ejecuta` la cuál contiene lo siguiente:
  - Atributos para representar : la línea de comando (el nombre del Ejecutador (`''python''` o `java.exe` o `java`) + la cadena que se le pasaba a la constructora cuando se creaba un objeto de esta clase en `EjecutaJava` o `EjecutaPython`), el directorio de trabajo, la entrada al proceso (en formato cadena o como fichero de entrada), el tipo de salida (cadena o fichero de salida), la salida de error (en formato cadena o fichero de error)

- Métodos para especificar cada uno de los atributos.
- Método para ejecutar la aplicación. Se realizan los siguientes pasos:
  - Se crea un proceso mediante la clase `ProcessBuilder` al que se le pasa como parámetro la línea de comando.
  - Se configura el directorio de trabajo y las posibles redirecciones de entrada y de salida del proceso.
  - Se inicia el proceso y se ejecuta.
  - Se guardan la salida y la salida de error de la ejecución del proceso.

4.3.6. Manejo de ficheros

Para la implementación de los métodos relacionados con el manejo de ficheros se ha usado la librería `util.io` que ofrece Java. En particular, usamos la clase `File` de dicha librería.

- **Nombre de un fichero** : Se construye un objeto de la clase `File` pasándole la ruta de fichero y llamando al método `getName()` obtenemos el nombre de dicho fichero. Ejemplo :

  "c: \users \Hola.txt" devuelve *"Hola.txt"*

- **Extensión de un fichero** : Primero se obtiene el nombre del fichero usando el método anterior. A continuación recorremos el nombre del fichero hasta encontrar el último ".". La extensión del fichero es toda la parte de la cadena que hay a partir de ese último punto. Ejemplo:

  "c: \users \Hola.txt" devuelve *"txt"*

- **Eliminar extensión de un fichero** : Dada la cadena que representa el nombre del fichero (puede ser también la ruta) nos quedamos con la porción de cadena que va desde el principio de ésta hasta la aparición del último punto. Ejemplo:

  "c: \users \Hola.txt" devuelve *"c: \users \Hola"*

- **Contenido de un fichero** : Abrimos el fichero en modo lectura y lo vamos leyendo línea a línea y las vamos concatenando a una cadena de texto hasta que ya no haya más líneas.

- **Escribir una cadena de texto en un fichero** : Abrimos el fichero en modo escritura y añadimos la cadena al final del contenido del fichero.
- **Eliminar un fichero** : Se construye un objeto de la clase `File` pasándole la ruta del fichero y se llama al método `delete()` de dicha clase.
- **Comparar 2 ficheros** : Se abren los 2 ficheros en modo lectura y se van comparando línea a línea hasta que nos encontremos 1 línea distinta (se devuelve `false`) o hasta que se ha llegado al final de ambos ficheros a la vez (se devuelve `true`).
- **Concatenar contenido de 2 ficheros** : Si queremos anádir el contenido del fichero A al final del contenido del fichero B, abrimos el fichero A en modo lectura y el fichero B en modo escritura. A continuación, leemos el fichero A línea a línea y las vamos escribiendo al final del fichero B.

4.3.7. Manejo de directorios

Para la implementación de los métodos relacionados con el manejo de directorios se ha usado la librería `util.io` que ofrece Java. En particular, usamos la clase `File` de dicha librería.

Para explicar cada uno de los métodos, vamos a usar como ejemplo el árbol de directorios y ficheros de la figura 4.4:

- **Ficheros de un directorio** : Recorremos el contenido de un directorio, si se encuentra un fichero guarda su nombre en una lista y si es un directorio no hace nada. Cuando ha recorrido todo el directorio, devolvemos la lista de los nombres de los ficheros. Ejemplo :

  Ficheros de *"home \Rosa"* son *"tareas.txt"*

- **Ficheros de un directorio (búsqueda recursiva)** : Recorremos el contenido de un directorio, si se encuentra un fichero guarda su nombre en una lista y si es un directorio busca los ficheros de ese subdirectorio. Cuando se ha recorrido todo el directorio y sus subdirectorios, devolvemos la lista de nombres de ficheros. Ejemplo:

  Ficheros de *"home \Rosa"* son *"tareas.txt,enero.txt" y "febrero.txt"*

- **Subdirectorios de un directorio** : Recorre el contenido de un directorio, si se encuentra un directorio guarda su nombre en una lista y

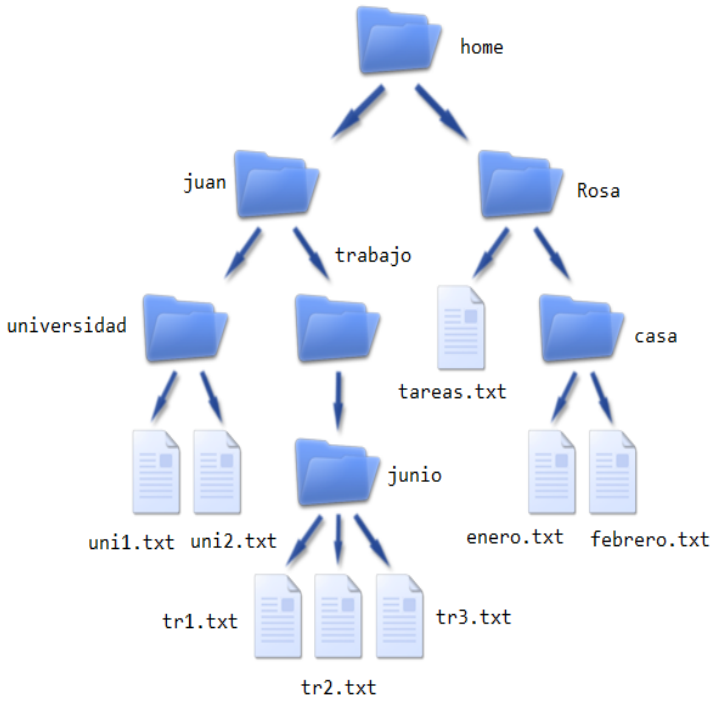


Figura 4.4: Ejemplo de árbol de directorios

si es un fichero no hace nada. Cuando ha recorrido todo el directorio, devuelve la lista de los nombres de los subdirectorios. Ejemplo:

Los subdirectorios del directorio *"home"* son *"home \juan"* y *"home \Rosa"*

- **Número de ficheros de un directorio** : Inicializa una variable a 0 ,recorremos el contenido del directorio y por cada fichero que nos encontremos sumamos 1 a la variable. Cuando hemos recorrido todo el directorio, devolvemos el valor de la variable. Ejemplo :

  "home \Rosa" tiene 1 fichero (*"tareas.txt"*)

- **Crear directorio temporal** : Buscamos el directorio temporal del sistema operativo en el que nos encontramos y después intentamos crear hay un nuevo directorio y, si se ha podido crear, devolvemos la ruta de dicho directorio.
- **Eliminar un directorio** : Primero se recorre el directorio buscando todos sus subdirectorios y borrándolos uno por uno. Cuando se han

borrado todos los subdirectorios, se borra el directorio. Ejemplo :

Imaginemos que queremos eliminar el directorio *"home \juan"* primero hay que eliminar el directorio *"home \juan \universidad"*, después eliminar el directorio *"home \juan \trabajo"* (antes hay que eliminar el directorio *"home \juan \trabajo \junio"*) y por último ya se puede eliminar el directorio *"home \juan"*.

- **Crear un directorio** : Se crea el directorio que se pasa como parámetro creando alguno de sus antecesores si es necesario. Esto último quiere decir que si queremos crear el directorio `c: \home \local` y no existe el directorio `c: \home` pues se crea antes.

- **Comprobar la existencia de un directorio** : Para comprobar si existe un directorio, se crea un objeto de la clase File pasándole como parámetro el directorio y se llama al método `isDirectory()` de dicha clase. Se usa el método `isDirectory()` porque puede existir un fichero con el mismo nombre que el directorio que se busca.

- **Crear ruta a partir de rutas parciales** : Dada una lista de cadenas que representan rutas parciales las concatenamos para formar una única ruta. Ejemplo :

  A partir de *home,local,universidad* se crea la ruta *"home \local \universidad"* si se está en Windows y sino se crea la ruta *"home/local/universidad"*. .

4.3.8. Manejo de ficheros zip

Para descomprimir un fichero zip se utiliza la clase `zip` de la librería `java.util.zip`.

Dado un archivo zip A y un directorio de destino B se realizan los siguientes pasos:

1. Se crea un objeto de la clase `ZipInputStream` al que se le pasa como parámetro A.

2. Por cada entrada de A, si esa entrada es un directorio se crea un directorio con ese nombre en el directorio de salida. Si no es un directorio, se crea en el directorio de destino un fichero con el mismo nombre y se copia en él el contenido del fichero (la entrada de A).

4.4. Nodos Avanzados

En esta sección vamos a usar los nodos creados anteriormente para crear nodos más avanzados.

4.4.1. RecoverFromFatalError

Nodo que tiene un nodo hijo que ejecuta. Si falla el nodo hijo, el nodo fallará si la variable `Critical Error` almacenada en el contexto tiene valor `true`. Se ejecuta de la siguiente manera :

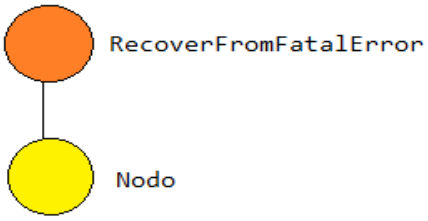


Figura 4.5: Nodo RecoverFromFatalError

- Se ejecuta el nodo hijo. Si la ejecución tiene éxito (si devuelve COMPLETED), devuelve COMPLETED.
- Si no se ha ejecutado con éxito se consulta el valor de la variable `Critical Error` del contexto. Si el valor es `true` devuelve FAILED y si no, devuelve COMPLETED.

4.4.2. ExisteFicheroOCriticalError

Comprueba si existe un fichero y si no existe, genera un error crítico. Tiene un campo para indicar el fichero que se quiere ver si existe o no.

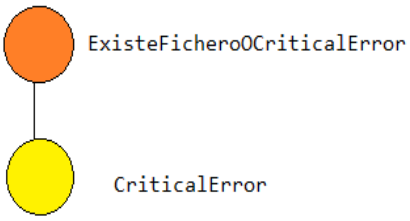


Figura 4.6: Nodo ExisteFicheroOCriticalError

Se ejecuta de la siguiente manera :

- Si existe el fichero, devuelve COMPLETED.

- Si no existe, ejecuta el nodo `CriticalError` el cuál muestra un mensaje informando de que el fichero no existe, y devuelve FAILED.

4.4.3. ExisteDirectorioOCriticalError

Comprueba si existe un directorio y si no existe, genera un error crítico. Tiene un campo para indicar el directorio que se quiere ver si existe o no.

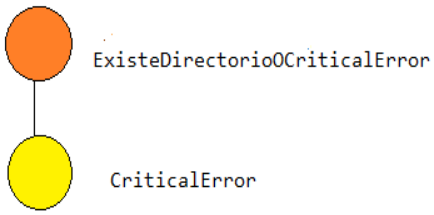


Figura 4.7: Nodo ExisteDirectorioOCriticalError

Se ejecuta de la siguiente manera :

- Si existe el directorio, devuelve COMPLETED.
- Si no existe, ejecuta el nodo `CriticalError` el cuál muestra un mensaje informando de que el directorio no existe, y devuelve FAILED.

4.4.4. ExecuteOrFatalError

Ejecuta el nodo que se le pasa como parámetro o genera un error fatal. Se ejecuta de la siguiente manera:

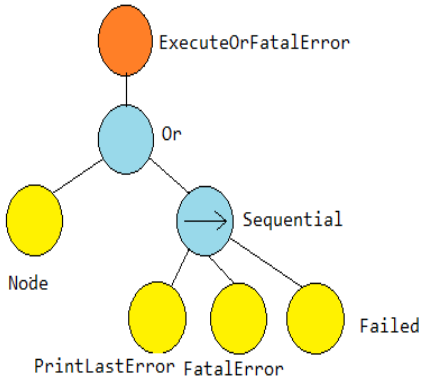


Figura 4.8: Nodo ExecuteOrFatalError

- Ejecuta el nodo `Or` que, aunque ya lo hemos explicado anteriormente, vamos a recordarlo:

  - Ejecuta el nodo `Node`. Si la ejecución tiene éxito, devuelve COMPLETED.
  - Si ha fallado, ejecuta el nodo `Sequential` el cuál primero muestra el valor de la variable `Last Error` del contexto mediante el nodo `PrintLastError`, después muestra un error fatal que se le pasa por parámetro mediante el nodo `FatalError` y por último devuelve FAILED mediante el nodo `Failed`.

- Devuelve el resultado de la ejecución del nodo `Or`

4.4.5. ExecuteOrCriticalError

Ejecuta el nodo que se le pasa como parámetro o genera un error crítico. Se ejecuta de la siguiente manera:

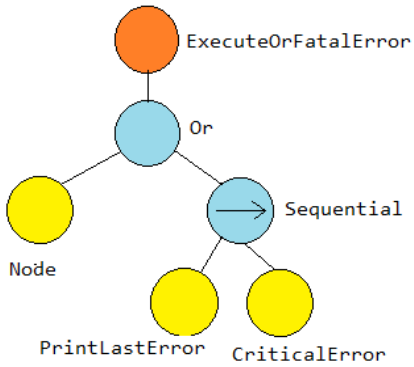


Figura 4.9: Nodo ExecuteOrCriticalError

- Ejecuta el nodo `Or` que, aunque ya lo hemos explicado anteriormente, vamos a recordarlo:
  - Ejecuta el nodo `Node`. Si la ejecución tiene éxito, devuelve COMPLETED.
  - Si ha fallado, ejecuta el nodo `Sequential` el cuál primero muestra el valor de la variable `Last Error` del contexto mediante el nodo `PrintLastError` y después muestra un error cr'tico que se le pasa por parámetro mediante el nodo `CriticalError`. En este caso, siempre devuelve FAILED.
- Devuelve el resultado de la ejecución del nodo `Or`

4.4.6. ExecuteOrCriticalError

Ejecuta el nodo que se le pasa como parámetro o genera un error crítico. Se ejecuta de la siguiente manera:

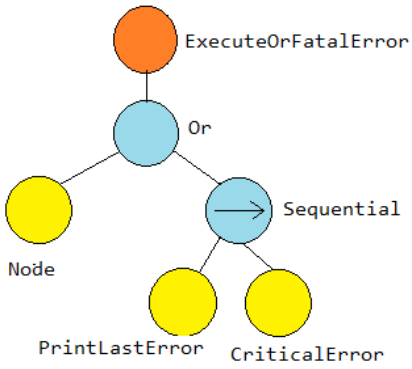


Figura 4.10: Nodo ExecuteOrCriticalError

- Ejecuta el nodo `Or` que, aunque ya lo hemos explicado anteriormente, vamos a recordarlo:
  - Ejecuta el nodo `Node`. Si la ejecución tiene éxito, devuelve COMPLETED.
  - Si ha fallado, ejecuta el nodo `Sequential` el cuál primero muestra el valor de la variable `Last Error` del contexto mediante el nodo `PrintLastError` y después muestra un error cr'tico que se le pasa por parámetro mediante el nodo `CriticalError`. En este caso, siempre devuelve FAILED.
- Devuelve el resultado de la ejecución del nodo `Or`

4.4.7. EjecutaAplicacion

Ejecuta una aplicación previamente compilada. No se muestra la estructura del nodo porque no tiene nodos hijo.

Tiene como atributos el nombre del fichero ejecutable, el directorio de trabajo, el fichero de entrada y el de salida, y la lista de parámetros.

Contiene los siguientes métodos:

- Métodos para especificar los valores de cada uno de los atributos.
- Un método que ejecuta el nodo. Realiza los siguientes pasos:
  - Si el fichero ejecutable tiene extensión `.class`, llama al método que ejecuta código Java y devuelve si se ha podido ejecutar con éxito.
  - Si el fichero ejecutable tiene extensión `.py`, llama al método que ejecuta código Python y devuelve si se ha podido ejecutar con éxito.

    - En otro caso, llama al método que ejecuta código nativo y devuelve si se ha podido ejecutar con éxito.

- Un método para ejecutar código Java. Realiza los siguientes pasos:
    - Crea un objeto de tipo `EjecutaJava` y le especifica los siguientes valores : nombre de la clase principal (fichero ejecutable eliminándole la extensión), el directorio de trabajo, la lista de parámetros, el directorio con los `.jar` necesarios, el directorio con los .class necesarios y los ficheros de entrada y salida.
    - Llama al método `ejecuta()` de dicho objeto.
    - Almacena en el contexto una variable `Last Exit Code` con el valor devuelto por la ejecución.
    - Devuelve COMPLETED.
- Un método para ejecutar código Python. Realiza los siguientes pasos:
    - Crea un objeto de tipo `EjecutaPython` y le especifica los siguientes valores : nombre del fichero ejecutable y la lista de parámetros.
    - Llama al método `ejecuta()` de dicho objeto.
    - Almacena en el contexto una variable `Last Exit Code` con el valor devuelto por la ejecución.
    - Devuelve COMPLETED.
- Un método para ejecutar código nativo. Realiza los siguientes pasos:
    - Crea un objeto de tipo `Ejecuta` y le especifica los siguientes valores : el fichero ejecutable, la lista de parámetros, el directorio de trabajo y los ficheros de entrada y salida.
    - Llama al método `ejecuta()` de dicho objeto.
    - Almacena en el contexto una variable `Last Exit Code` con el valor devuelto por la ejecución.
    - Devuelve COMPLETED.

4.4.8. CompilaFuente

Compila un fichero de código fuente pasado como parámetro. No se muestra la estructura del nodo porque no tiene nodos hijo.

Tiene como atributos el directorio de salida, la ruta del fichero fuente, el nombre del fichero fuente, el nombre del fichero ejecutable de salida y la extensión del fichero fuente.

Contiene los siguientes métodos:

- Una constructora que recibe como parámetros el directorio de salida y el nombre completo del fichero fuente, y obtiene a partir de estos el valor de los atributos del nodo.
- Un método que comprueba si el fichero fuente está escrito en alguno de los lenguajes de programación admitidos (Java,C,CPP).
- Un método que ejecuta el nodo. Realiza los siguientes pasos:
  - Si el fichero fuente está escrito en C, llama al método que compila código en C y devuelve si se ha podido compilar con éxito.
  - Si el fichero fuente está escrito en CPP, llama al método que compila código en CPP y devuelve si se ha podido compilar con éxito.
  - Si el fichero fuente está escrito en Java, llama al método que compila código en Java y devuelve si se ha podido compilar con éxito.
  - En otro caso, almacena en el contexto una variable `Last Error` cuyo valor es un mensaje indicando que ese código fuente por el tipo de extensión no está permitido.
- Un método encargado de compilar código C. Realiza los siguientes pasos:
  - Crea un objeto de la clase `CCompilation` y le asigna el compilador de C (si no encuentra un compilador de C en el sistema devuelve FAILED).
  - Añade a dicho objeto el nombre del fichero fuente y el fichero ejecutable de salida.
  - Llama al método `compile()` de dicho objeto. Si la compilación tiene éxito, guarda en el contexto la variable `Last Executable` con valor la ruta del fichero ejecutable de salida y devuelve COMPLETED.
  - En caso contrario, guarda en el contexto la variable `Last Error` con valor el error que se produjo cuando se intentó compilar y devuelve FAILED.
- Un método encargado de compilar código CPP. Realiza los siguientes pasos:
  - Crea un objeto de la clase `CPPCompilation` y le asigna el compilador de CPP (si no encuentra un compilador de CPP en el sistema devuelve FAILED).
  - Añade a dicho objeto el nombre del fichero fuente y el fichero ejecutable de salida.

  - Llama al método `compile()` de dicho objeto. Si la compilación tiene éxito, guarda en el contexto la variable `Last Executable` con valor la ruta del fichero ejecutable de salida y devuelve COMPLETED.
  - En caso contrario, guarda en el contexto la variable `Last Error` con valor el error que se produjo cuando se intentó compilar y devuelve FAILED.

- Un método encargado de compilar código Java. Realiza los siguientes pasos:

  - Crea un objeto de la clase `JavaCompilation` y le asigna el compilador de Java (si no encuentra un compilador de Java en el sistema devuelve FAILED).
  - Añade a dicho objeto el nombre del fichero fuente y el directorio de salida.
  - Llama al método `compile()` de dicho objeto. Si la compilación tiene éxito, guarda en el contexto la variable `Last Executable` con valor la cadena formada por la concatenación del directorio de salida, el nombre del fichero ejecutable de salida y ".class". Por último, devuelve COMPLETED.
  - En caso contrario, guarda en el contexto la variable `Last Error` con valor el error que se produjo cuando se intentó compilar y devuelve FAILED.

Capítulo 5

Ejemplo de Uso

Resumen: En esta tercera parte del proyecto, vamos a hablar del proceso de creación de la aplicación encargada del proceso de validación de prácticas. Para ello,nos centraremos en una práctica real en particular. En la sección 5.1 hablaremos de que va la práctica sobre la que se va a usar la aplicación. En la sección 5.2, hablaremos de todo el proceso de creación del árbol de comportamiento necesario para la validación de la práctica. Por último, en la sección 5.3, veremos el resultado de ejecución de dicho árbol de comportamiento.

5.1. Explicación de la práctica que se va a validar

En esta sección vamos a hablar de la práctica que se ha usado para comprobar el perfecto funcionamiento tanto de la aplicación como del framework desarrolados.

La práctica que se va a explicar fue mandada a los estudiantes de la asignatura `Tecnología de la Programación` de la carrera `DOBLE GRADO EN INGENIERÍA INFORMÁTICA - MATEMÁTICAS`.

La práctica es una aventura conversacional en la que el jugador se tendrá que mover por una serie de habitaciones hasta llegar a la salida. El jugador podrá realizar las siguientes acciones :

- Moverse a la habitación del Norte, a la del Sur, a la del Este o a la del Oeste. A veces, no podrá moverse en alguna de las direcciones.
- Mostrar un mensaje de ayuda.
- Abandonar la aventura.
- Soltar un objeto que se lleve en el inventario.

- Examinar la habitación donde se encuentra.
- Coger un objeto que haya en la habiación donde se encuentra.
- Usar un objeto del inventario.

La aventura se puede jugar desde la ventana de comandos donde el jugador tendrá que teclear las acciones que quiere ir realizando a lo largo de la partida y donde le irán apareciendo los mensajes de información de la aventura.

La aventura también se puede jugar a través de la interfaz gráfica que se muestra en la figura 5.1:

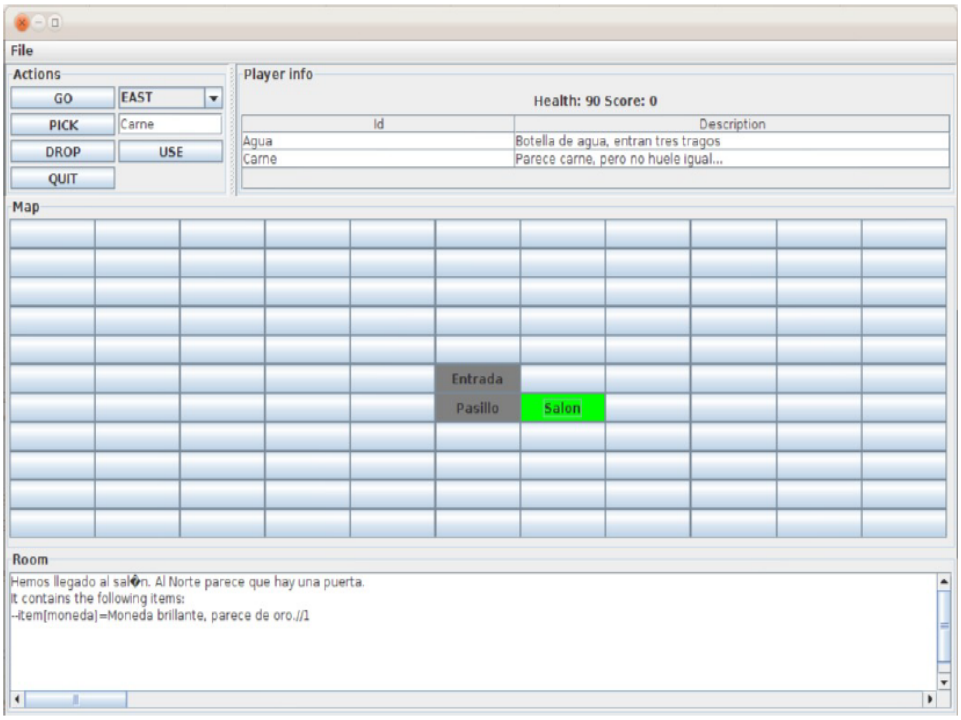


Figura 5.1: Interfaz de la aventura gráfica

5.2. Proceso de creación del árbol de comportamiento

El proceso de validación de la práctica consta de las siguientes fases:

- **Fase 1 : Comprobación preliminar del fichero a analizar** : Consta de los siguientes pasos:

  - Comprueba que el fichero existe.
  - Comprueba que el nombre del fichero sea de la forma `GrupoXX.zip` (XX = número de 2 cifras).
  - Extrae el número del grupo a partir del nombre del fichero (XX).
  - Comprueba que el fichero zip ocupa menos de 100Kb.

- **Fase 2 : Comprobación y configuración del entorno de validación** : Consta de los siguientes pasos:

  - Comprueba que hay un compilador Java disponible.
  - En sistemas que no sean Windows, comprueba que existe Python.
  - Crea un directorio temporal de trabajo.
  - Descomprime en el directorio de trabajo el fichero zip.

- **Fase 3 : Comprobación básica del fichero entregado después de descomprimirlo** : Consta de los siguientes pasos:

  - Comprueba la existencia del archivo de texto `alumnos.txt`.
  - Lee su contenido y lo guarda.
  - Comprueba que no existe ningún fichero .class.
  - Si existen, los borra.
  - Comprueba que no existen los directorios `testProfesor` y `bin`. Si existen, los borra.
  - Comprueba que existe el directorio `src`.

- **Fase 4 : Compilación de la práctica** : Consta de los siguientes pasos:

  - Crea el directorio `bin`.
  - Instala `jargs.jar` en el directorio de trabajo.
  - Compila la práctica colocando los .class en el directorio `bin`.

- **Fase 5 : Test de unidad** : Consta de los siguientes pasos:

  - Descomprime el fichero de test incluido en la aplicación (`TestPr5.zip`) en el directorio de trabajo.
  - Instalar `junit` en el directorio de trabajo.
  - Compilar los tests en el directorio `binTest`.
  - Los ejecuta y comprueba que pasan todos.

- **Fase 6 : Ejecución de la práctica** : Consta de los siguientes pasos:

- Copia al directorio de trabajo los ficheros que se usarán como entrada a la práctica.
- Instala el comparador de ficheros.
- Ejecuta la práctica varias veces con distintos parámetros y va acumulando la salida en un fichero de texto.
- Compara esa salida con la salida esperada.

A continuación, vamos a explicar la creación de los árboles de comportamiento encargados de ejecutar cada una de las fases anteriores. En la explicación de los árboles y de los nodos, no vamos a hablar de los nodos encargados de informar de la fase de validación por la que vamos ni tampoco por el paso por el que vamos (tampoco vamos a mostrarlos en las figuras) para no complicar ni las figuras ni las explicaciones.

5.2.1. Fase 1 : Comprobación preliminar del fichero a analizar

El árbol a ejecutar es el de la figura 5.2 (se ejecuta de arriba a abajo).

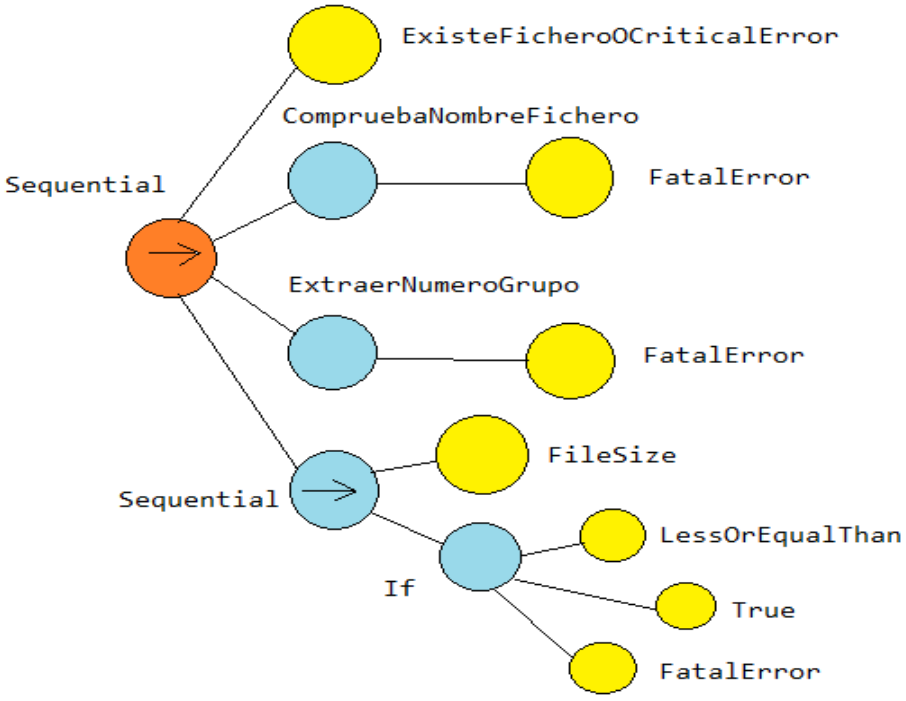


Figura 5.2: Árbol de la fase 1

El nodo `ExisteFicheroOCriticalError` comprueba si existe el fichero

zip. Devuelve COMPLETED si existe y FAILED si no existe.

El nodo `CompruebaNombreFichero` comprueba que el nombre del fichero zip es de la forma GrupoXX.zip. Si no lo es, ejecuta el nodo `FatalError` el cuál informa de que el nombre del fichero es incorrecto y devuelve FAILED. Si lo es, devuelve COMPLETED.

El nodo `ExtraeNumeroGrupo` intenta extraer el número del grupo a partir del nombre del fichero zip. Si no existe el número o no tiene 2 cifras, ejecuta el nodo `FatalError` el cuál informa de que no hay número de grupo o es incorrecto, y devuelve FAILED. En otro caso, almacena el número en una variable `Numero Grupo` del contexto y devuelve COMPLETED.

El nodo `FileSize` cálcula el tamaño del fichero zip y lo almacena en la variable `File Size` del contexto. Devuelve siempre COMPLETED.

El nodo `If` mira si el tamaño del fichero zip (valor de la variable `File Size` del contexto) es menor o igual que el tamaño máximo permitido (valor de la variable `Max Size` del contexto) y si lo es, ejecuta el nodo `True` el cuál devuelve COMPLETED. Si no, se ejecuta el nodo `FatalError` el cuál informa de que el fichero zip no tiene un tamaño permitido y devuelve FAILED.

A este árbol le vamos a llamar nodo `ComprobacionPreliminarFichero`.

5.2.2. Fase 2 : Comprobación y configuración del entorno de validación

El árbol a ejecutar es el de la figura 5.3 (se ejecuta de arriba a abajo).

El nodo `BuscaCompiladorJava` se encarga de buscar el compilador de Java del sistema y si lo encuentra, ejecuta el nodo `PrintAction` el cuál informa de la versión del compilador y devuelve COMPLETED. Si no lo encuentra, ejecuta el nodo `FatalError` el cuál informa de que no se ha encontrado ningún compilador de Java y devuelve FAILED.

El nodo `CompruebaPython` se encarga de buscar Python si no se está en Windows. Si se está en Windows devuelve COMPLETED. Si no, busca Python en el sistema y si lo encuentra se ejecuta el nodo `PrintAction` el cuál informa de la versión de Python y devuelve COMPLETED. Si no, ejecuta el nodo `FatalError` el cuál informa de que no se ha podido encontrar Python en el sistema y devuelve FAILED.

El nodo `CrearDirectorioTemporal` se encarga de crear un directorio

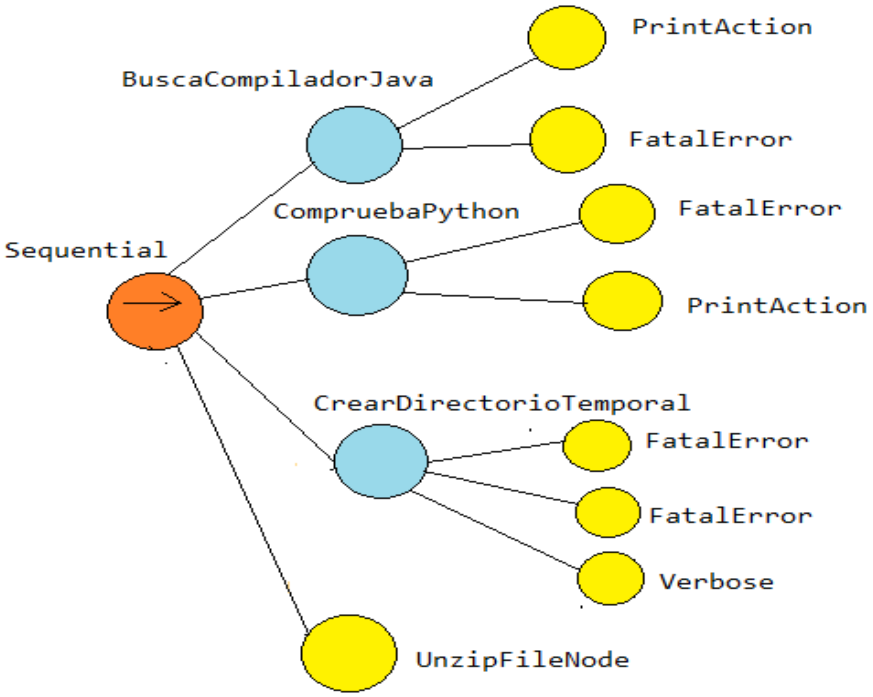


Figura 5.3: Árbol de la fase 2

temporal de trabajo. Si la variable `debugMode` del contexto tiene valor `true`, se realizan los siguientes pasos:

- Mira si existe el directorio ‘‘`temp`’’. Si existe y no lo puede borrar‘, ejecuta un nodo `FatalError` el cuál informa de que no se ha podido borrar y devuelve FAILED.
- Intenta crear el directorio ‘‘`temp`’’ y lo añade como valor a la variable `Working Dir` del contexto. Si no lo ha podido crear, ejecuta un nodo `FatalError` el cuál informa de que no se ha podido crear el directorio de trabajo y devuelve FAILED. Si lo ha podido crear, devuelve COMPLETED.

Si la variable `debugMode` del contexto tiene valor `false`, crea un directorio temporal. Si se ha podido crear, asigna la ruta de ese directorio a la variable `Working Dir` del contexto y devuelve COMPLETED. Si no, ejecuta un nodo `FatalError` el cuál informa de que no se ha podido crear y devuelve FAILED.

El nodo `UnzipFileNode` descomprime el fichero zip en el directorio de trabajo. Si lo consigue descomprimir, devuelve COMPLETED y si no, devuelve FAILED.

A este árbol le vamos a llamar nodo `MontaEntornoValidacion`.

5.2.3. Fase 3 : Comprobación básica del fichero entregado (después de descomprimirlo)

El árbol a ejecutar es el de la figura 5.4 (se ejecuta de arriba a abajo).

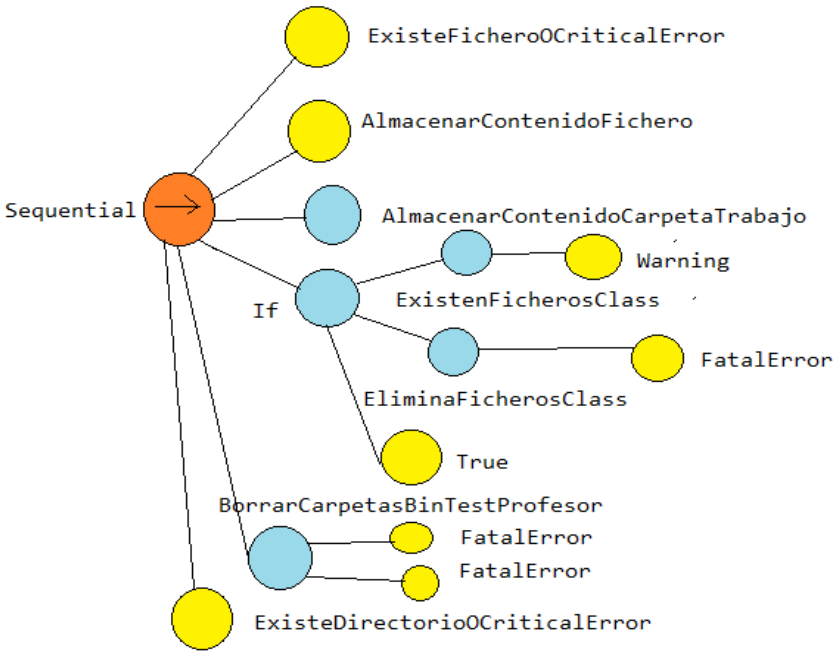


Figura 5.4: Árbol de la fase 3

El nodo `ExisteFicheroOCriticalError` se encarga de comprobar si existe el fichero `alumnos.txt`. Si existe devuelve FAILED y si no, devuelve COMPLETED

El nodo `AlmacenarContenidoFichero` almacena el contenido del fichero `alumnos.txt` en la variable `Contenido Fichero` del contexto. Siempre devuelve COMPLETED.

El nodo `If` comprueba si existen ficheros `.class` y si no existen, devuelve COMPLETED. Si existen, ejecuta el nodo `Warning` el cuál informa de que la próxima vez hay que borrar los `.class` y después ejecuta el nodo `EliminaFicherosClass`. Este nodo intenta borrar todos los ficheros `.class` y si lo consigue, devuelve COMPLETED. En otro caso, se ejecuta el nodo `FatalError` el cuál se informa de que no se han podido borrar, y devuelve

FAILED.

El nodo `BorrarCarpetasBinTestProfesor` comprueba si existen las carpetas `bin` y `testProfesor`. Si existen, ejecuta los nodos `Warning` los cuáles informan de que la próxima vez no se incluyan esas carpetas, e intenta borrarlas. Si no lo consigue, devuelve FAILED. Después de borrarlas, en el caso de que existieran, devuelve COMPLETED.

El nodo `ExisteDirectorioOCriticalError` comprueba si existe el directorio `src`. Si no Existe devuelve FAILED y si existe, devuelve COMPLETED

A este árbol le vamos a llamar nodo `ComprobacionBasicaFichero`.

5.2.4. Fase 4 : Compilación de la práctica

El árbol a ejecutar es el de la figura 5.5 (se ejecuta de arriba a abajo).

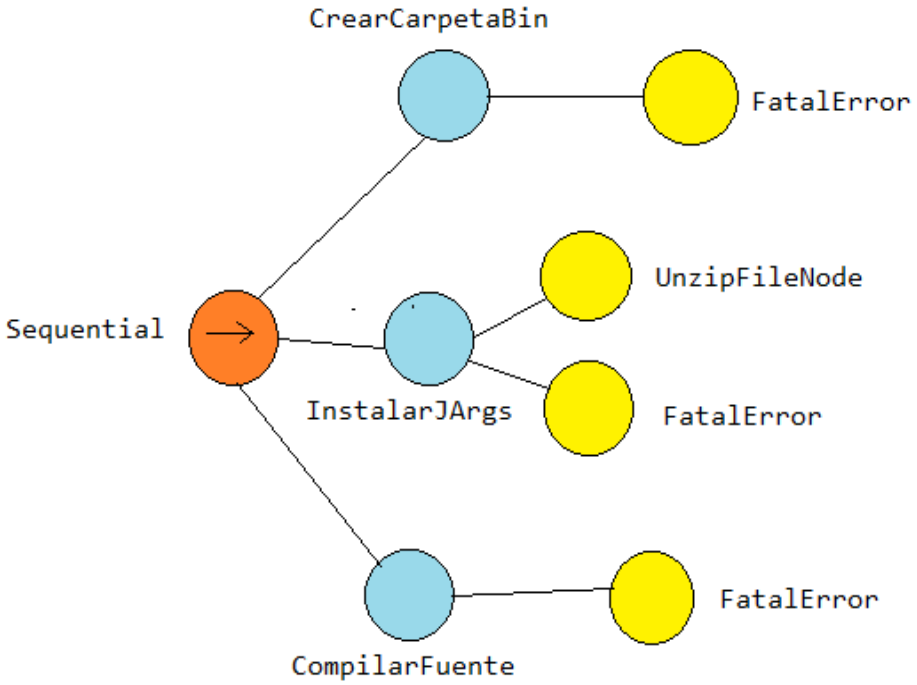


Figura 5.5: Árbol de la fase 4

El nodo `CrearCarpetaBin` intenta crear la carpeta `bin` en el directorio de trabajo. Si no se ha podido crear, se ejecuta el nodo `FatalError` el cuál informa de que no se ha podido crear, y devuelve FAILED. En caso contrario, devuelve COMPLETED.

El nodo `InstalarJArgs` intenta instalar `jargs.jar` en el directorio de trabajo. Para ello primero ejecuta el nodo `UnzipFileNode` el cuál intenta descomprimir `jargs.zip` en el directorio de trabajo. Si no se ha podido descomprimir, ejecuta el nodo `FatalError` el cuál informa de que no se ha podido descomprimir, y devuelve FAILED. En caso contrario, devuelve COMPLETED.

El nodo `CompilarFuente` intenta compilar los ficheros fuente de la práctica. Para ello, crea un objeto de tipo `JavaCompilation` al que se le pasa la ruta de la clase Main.java, la ruta del `jargs.jar` y las rutas de las carpetas `src` y `bin`. Si la llamada al método `compile()` de dicho objeto tiene éxito, guarda en la variable `Practica Ejecutable` del contexto la ruta del Main.class y devuelve COMPLETED. Si no ha tenido éxito, ejecuta el nodo `FatalError` el cuál informa de que no se ha podido compilar la práctica y devuelve FAILED.

A este árbol le vamos a llamar nodo `CompilarPractica`.

5.2.5. Fase 5 : Test de unidad

El árbol a ejecutar es el de la figura 5.6 (se ejecuta de arriba a abajo).

El nodo `DescomprimeValidador` se encarga de descomprimir el fichero `validador.zip`, que se entrega junto a la aplicación, en el directorio de trabajo. Esto lo realiza ejecutando el nodo `UnzipFileNode`. Si no se ha podido descomprimir, ejecuta el nodo `FatalError` el cuál informa de que no se ha podido descomprimir, y devuelve FAILED. En caso contrario, devuelve COMPLETED.

El nodo `DescomprimeTests` se encarga de descomprimir el fichero `TestPr5.zip`, que se entrega junto a la aplicación, en el directorio de trabajo. Esto lo realiza ejecutando el nodo `UnzipFileNode`. Si no se ha podido descomprimir, ejecuta el nodo `FatalError` el cuál informa de que no se ha podido descomprimir, y devuelve FAILED. En caso contrario, devuelve COMPLETED.

El nodo `InstalaJUnit` intenta instalar `junit.jar` en el directorio de trabajo. Para ello primero ejecuta el nodo `UnzipFileNode` el cuál intenta descomprimir `junit.zip` en el directorio de trabajo. Si no se ha podido descomprimir, ejecuta el nodo `FatalError` el cuál informa de que no se ha podido descomprimir.

El nodo `CrearCarpetaBinTest` se encarga de crear la carpeta `binTest` en

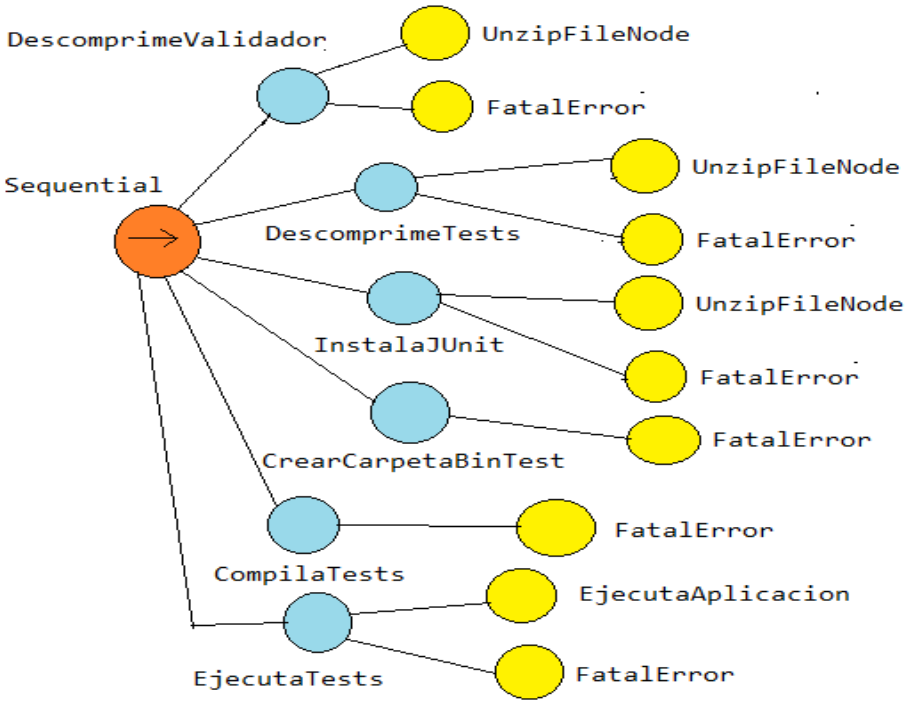


Figura 5.6: Árbol de la fase 5

el directorio de trabajo. Si no se puede crear, se ejecuta el nodo `FatalError` el cuál informa de que no se ha podido crear, y devuelve FAILED. En caso contrario, devuelve COMPLETED.

El nodo `CompilaTests` se encarga de compilar los tests. Para ello, crea un objeto de tipo `JavaCompilation` al que se le pasa la ruta de la clase AllTests.java, la ruta del `junit.jar` y las rutas de las carpetas `src`,`testProfesor` y `binTest`. Si la llamada al método `compile()` de dicho objeto tiene éxito, guarda en la variable `Test Ejecutable` del contexto la ruta del AllTests.class y devuelve COMPLETED. Si no ha tenido éxito, ejecuta el nodo `FatalError` el cuál informa de que no se ha podido compilar los tests y devuelve FAILED.

El nodo `EjecutaTests` se encarga de ejecutar los tests. Para ello ejecuta el nodo `EjecutaAplicacion` al que se le pasa la ruta del `junit.jar`, la ruta de la carpeta `binTest`, `org.junit.runner.JUnitCore.class` como fichero ejecutable y como parámetro el valor de la variable `Test Ejecutable` del contexto. Si la ejecución tiene éxito, se devuelve COMPLETED y si no, se ejecuta el nodo `FatalError` el cuál informa de que no se ha podido ejecutar y devuelve FAILED.

A este árbol le vamos a llamar nodo `TestUnidad`.

5.2.6. Fase 6 : Ejecución de la práctica

El árbol a ejecutar es el de la figura 5.7 (se ejecuta de arriba a abajo).

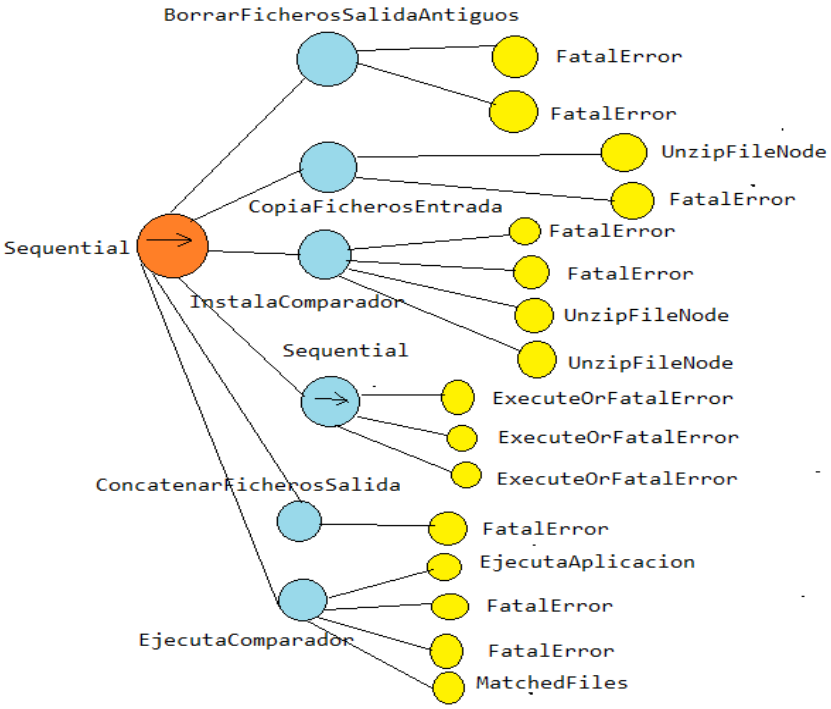


Figura 5.7: Árbol de la fase 6

El nodo `BorrarFicherosSalidaAntiguos` borra del directorio de salida, si existen, los ficheros `out.txt` y `salidaComparador.txt`. Si no los consigue borrar, ejecuta los nodos `FatalError` que informan de que no se han podido eliminar dichos ficheros, y devuelve FAILED. En caso contrario, devuelve COMPLETED.

El nodo `CopiaFicheroEntrada` copia los ficheros de entrada a la práctica en el directorio de trabajo. Para ello, descomprime mediante el nodo `UnzipFileNode` el fichero `EjemploMapaValidador.zip`. Si no se ha podido descomprimir, ejecuta el nodo `FatalError` el cuál informa de que no se ha podido descomprimir dicho fichero,y devuelve FAILED. En caso contrario, devuelve COMPLETED.

El nodo `InstalaComparador` descomprime el comparador de ficheros de texto en el directorio de trabajo.

Si se está en Windows, se intenta descomprimir el fichero `comparador.zip` en el directorio de trabajo. Si no se ha podido, se ejecuta el nodo `FatalError` el cuál informa de que no se ha podido descomprimir, y devuelve FAILED. En caso contrario, devuelve COMPLETED.

Si no se está en Windows, se intenta descomprimir el fichero `ComparadorPy.zip` en el directorio de trabajo. Si no se ha podido, se ejecuta el nodo `FatalError` el cuál informa de que no se ha podido descomprimir, y devuelve FAILED. En caso contrario, devuelve COMPLETED.

A continuación aparecen 3 nodos `ExecuteOrFatalError` que se encargan de ejecutar la práctica con distintos parámetros. A cada uno de esos nodos, se les pasa un nodo `EjecutaAplicacion`. A este último nodo se le pasa : la ruta del `jargs.jar`, la ruta de la carpeta `bin`, el valor de la variable `Practica Ejecutable` (ruta del `Main.class`) del contexto, la lista de parámetros y la ruta de los ficheros de entrada y salida.

El nodo `ConcatenarFicherosSalida` : concatena el contenido de los 3 ficheros de salida producidos por los nodos `EjecutaAplicación` en un único fichero `‘‘out.txt’’` que se encuentra en el directorio de salida. Devuelve COMPLETED si se han concatenado todos los ficheros de salida con éxito. En caso contraio, se ejecuta el nodo `FatalError` el cuál informa de que no se han podido concatenar, y devuelve FAILED.

El nodo `EjecutaComparador` se encarga de ejecutar el comparador de ficheros de texto. Realiza los siguientes pasos:

- Si se está en Windows, se le asigna al nodo `EjecutaAplicacion` : la ruta del `comparador.exe`, la lista de parámetros (la ruta del fichero `‘‘out.txt’’` y la ruta del fichero que contiene la salida esperada de la práctica si estuviera bien implementada) y la ruta del fichero de salida `‘‘salidaComparador.txt’’`. Si la ejecución de este nodo falla, se ejecuta el nodo `FatalError` el cuál indica que no se ha podido ejecutar y devuelve FAILED. En otro caso, ejecuta el nodo `MatchedFiles` el cuál indica cuánto se parece el fichero `‘‘out.txt’’` y el fichero `‘‘salidaEsperada.txt’’` (viene indicado por el contenido del fichero “salidaComparador.txt”). Por último, se devuelve COMPLETED.

- Si no se está en Windows, se configura el `System.out` para que redireccione la salida al fichero `‘‘salidaComparador.txt’’`. Después, se le asigna al nodo `EjecutaAplicacion` : la ruta del `compararTexto.py` y la lista de parámetros (la ruta del fichero `‘‘out.txt’’` y la ruta del fichero que contiene la salida esperada de la práctica si estuviera bien implementada `‘‘salidaEsperada.txt’’`). Si la ejecución de este no-

do falla, se ejecuta el nodo `FatalError` el cuál indica que no se ha podido ejecutar y devuelve FAILED. En otro caso, ejecuta el nodo `MatchedFiles` el cuál indica cuánto se parece el fichero `‘‘out.txt’’` y el fichero `‘‘salidaEsperada.txt’’` (viene indicado por el contenido del fichero “salidaComparador.txt”). Por último, se devuelve COMPLETED.

A este árbol le vamos a llamar nodo `EjecutaPractica`.

5.2.7. Árbol final de validación

El árbol final de validación es el que se muestra en la figura 5.8:

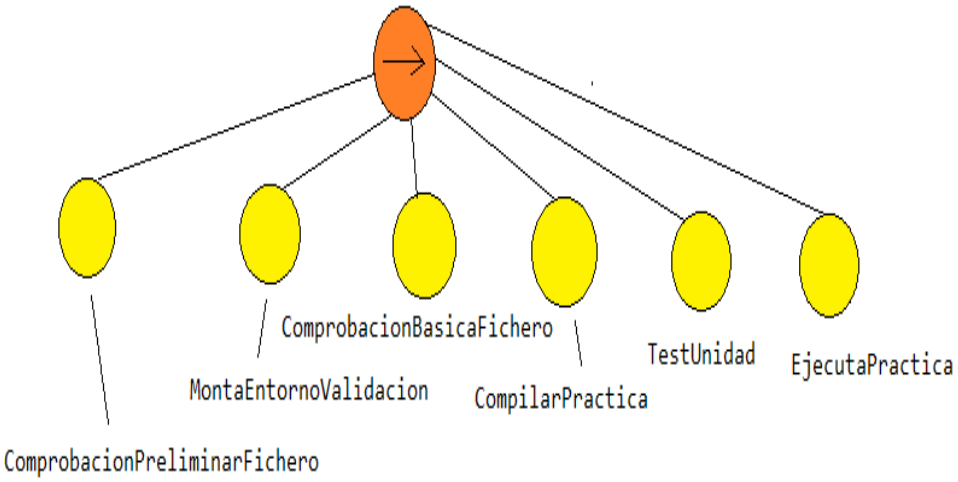


Figura 5.8: Árbol final de validación

5.3. Ejemplo de validación

En esta sección vamos a validar la práctica entregada por el alumno (la cuál está bien implementada). Para ello ejecutamos la plicación y obtenemos el resultado que se muestra en la figura 5.9.

Como podemos observar, la práctica está bien implementada (refiriendose a que no ha fallado la ejecución del árbol de validación) y que el fichero de salida que produce la práctica del alumno se parece en un 99.56 % al fichero de salida esperado.

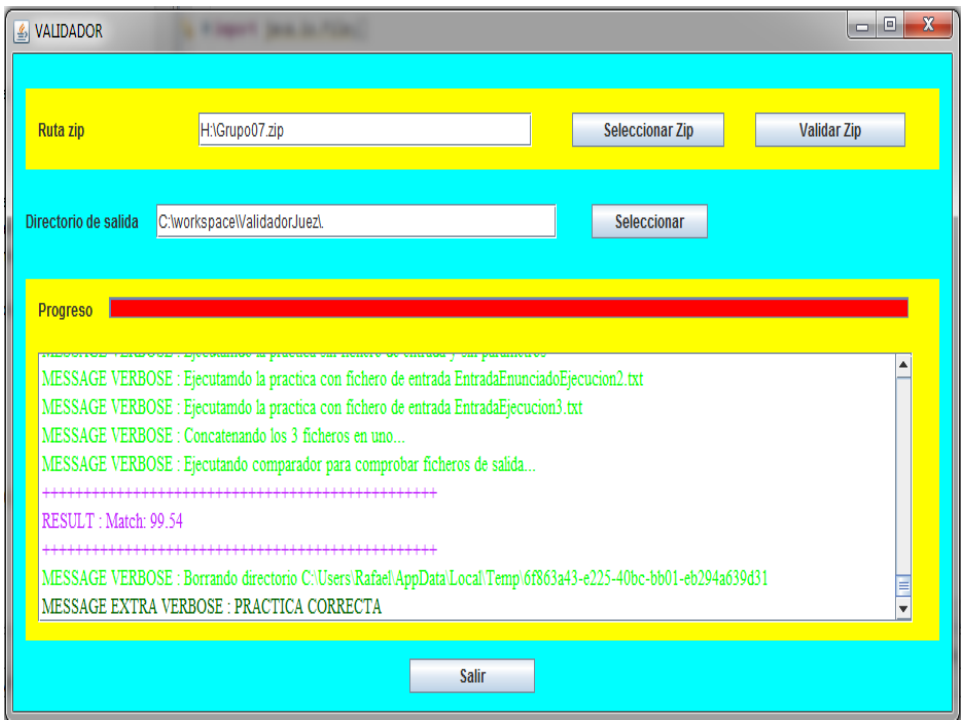


Figura 5.9: Resultado de la validación de la práctica

Capítulo 6

Trabajo futuro y conclusiones

Resumen: En la sección 6.1 hablaremos de las conclusiones a las que se han llegado después de desarrollar tanto la parte del framework como la parte de la aplicación encargada de la validación de prácticas. En la sección 6.2 hablaremos de las posibles mejoras que se le pueden realizar.

6.1. Conclusiones

6.1.1. Beneficios del uso de la aplicación

Tanto el framework como la aplicación se han desarrollado para funcionar tanto en Windows como en otros sistemas operativos por lo que cualquier estudiante o profesor podrá usarlo sin importar el sistema operativo que tenga instalado en su ordenador.

Gracias al desarrollo de ambas cosas, los profesores tardarán menos tiempo en corregir las prácticas de los alumnos ya que todo el proceso de validación de la práctica lo podrán realizar los estudiantes. Además, gracias a él, cuando los alumnos entreguen una práctica al profesor estarán seguros al 100 % de que la práctica funciona correctamente (ya que en caso contrario les avisará de la incorrecta implementación de la práctica).

6.1.2. Cosas aprendidas

Gracias al desarrollo de la framework, se ha aprendido todo lo relacionado con los árboles de comportamiento.

También ha ayudado a comprender el trabajo duro que tienen que realizar los profesores cuando tienen que comprobar la correcta implementación de

las prácticas de los alumnos al tener que realizar un montón de pruebas sobre cada práctica entregada para la comprobación de la validez de cada entrega.

6.2. Trabajo futuro

Tanto la parte del framework como la aplicación desarrollados pueden ser mejorada en varios aspectos. A continuación aparecen algunas de las mejoras que se pueden realizar:

- Uno de los problema es que, cada vez que se quiera ejecutar un árbol de comportamiento distinto habrá que modificar el código de la aplicación . Por lo tanto, una posible mejora sería que los árboles de comportamiento se pudieran crear mediante una interfaz gráfica de una manera parecida a como lo hace el framework JBT de Java tal y como se explicó en el Capítulo 2.
- Otra posible mejora podrá ser que la parte de la aplicación pudiera ser accesible a través de Internet como si fuera una aplicación web. De esta manera, el profesor no tendría que pasar a cada alumno el validador de prácticas.

Capítulo 7

Bibliografía

- http://www.starcostudios.com/blog/2010/02/arboles-de-comportamiento-behaviour-trees-para-gestionar-comportamientos/

- PALMA DURÁN, R. J. *Java Behaviour Trees, User Guide* , http://download.polytechnic.edu.na/pub4/download.sourceforge.net/pub/sourceforge/j/jb/jbt/UserGuide/UserGuide _ 0.0.2/UserGuide.pdf, *September 2, 2010*

- SHAMSI, F. y ELNAGAR, A. *An Intelligent Assessment Tool for Students Java Submissions in Introductory Programming Courses* Journal of Intelligent Learning Systems and Applications, Vol. 4 No. 1, 2012, pp. 59-69. Disponible en : *http://www.scirp.org/journal/PaperInformation.aspx?paperID=17557*

Apéndice A

Manual de uso de la aplicación y del framework

Resumen: En la sección A.1 se explicará los pasos que tiene que seguir un estudiante para usar la aplicación de validación de prácticas. En la sección A.2 se explicarán los pasos que tiene que seguir un profesor para usar tanto el framework como la aplicación desarrollado para la validación de prácticas.

A.1. Manual de profesor

Un profesor que quiera usar tanto el framework como la aplicación para ayudar a sus alumnos a que estos validen sus prácticas tendrá que hacer lo siguiente:

- Modificar la parte del framework creando todos los nuevos nodos básicos y avanzados que se necesiten.
- Modificar la parte de la aplicación creando todos los nuevos nodos que necesite para crear las etapas del proceso de validación de la práctica.
- Crear una nueva clase principal parecida a la clase `ValidaPractica5.java` que se incluye en la aplicación o modificar esta clase directamente. Lo único que tiene que modificar es la construcción del árbol de comportamiento.
- También puede, aunque no es necesario, cambiar el aspecto y/o funcionalidad de la interfaz gráfica creada en la clase `ValidatorWindow.java` para que se adapte a la práctica a entregar por los alumnos.

A.2. Manual de estudiante

Un estudiante que quiera usar la aplicación para validar su práctica tiene que seguir los siguientes pasos :

- Ejecutar la aplicación.
- Especificar la ruta del zip que contiene los ficheros fuente de la práctica implementada. Esto lo puede hacer bien escribiendo a mano la ruta en el campo de texto correspondiente o bien seleccionándolo a través del cuadro de diálogo que aparece al pulsar el botón *Seleccionar Zip*.
- Especificar el directorio de salida. Esto lo puede hacer bien escribiendo a mano la ruta en el campo de texto correspondiente o bien seleccionándolo a través del cuadro de diálogo que aparece al pulsar el botón *Seleccionar*.
- Pulsar el botón *Validar zip* (NOTA : si no ha especificado alguno de los campos anteriores, no empezará la ejecución y se le mostrará un mensaje de error).
- Estar atento a los posibles mensajes de aviso y/o error que le podrán aparecer en el cuadro de información.
- Estar atento al mensaje en el que se le indica cuánto se parece la solución que da su práctica a la solución oficial proporcionada por el profesor.
- Pulsar el botón *Salir* para salir de la aplicación.